UNIVERSIDAD AUTONOMA DE MADRID

Escuela Politécnica Superior

18|19

Trabajo fin de grado

Redes neuronales recurrentes y autómatas finitos deterministas

Christian Oliva Moya

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

excelencia Campus Internacional UAM CSIC

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Redes neuronales recurrentes y autómatas finitos deterministas

Inferencia de gramáticas regulares

Autor: Christian Oliva Moya
Tutor: Luis Fernando Lago Fernández

mayo 2019

Christian Oliva Moya
Redes neuronales recurrentes y autómatas finitos deterministas

Christian Oliva Moya
C\ Francisco Tomás y Valiente Nº 11

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Resumen

Se ha realizado un estudio de las principales técnicas de inferencia de gramáticas regulares, comparando algunos algoritmos clásicos de este campo de investigación, como son $RPNI$ y $LSTAR$, con las nuevas tecnologías que presentan el aprendizaje profundo y, concretamente, las redes neuronales recurrentes. Además, se proponen ciertas mejoras sobre la arquitectura de red de Elman para proporcionar una solución a los siguientes problemas: (1) la generalización de los sistemas dinámicos al enfrentarlos a una secuencia temporal en un problema de clasificación, (2) la sobre-regularización resultante al aplicar regularización L1 a esta arquitectura de red, (3) evitar las técnicas de cuantización del espacio interno de estados y (4) proporcionar una nueva herramienta para analizar el comportamiento interno de la red neuronal.

Con estas mejoras la red desarrollada aumenta considerablemente su interpretabilidad, demostrando, por un lado, que las neuronas de la capa oculta tienen una función concreta y determinante en la codificación de la solución del problema y, por otro lado, que esta nueva red puede estar comportándose internamente como un autómata finito determinista.

Palabras clave

Inferencia de gramáticas, Redes Neuronales Recurrentes, Autómatas Finitos Deterministas

Abstract

Deep learning and recurrent neural networks (RNN) applied to infer regular grammars have been compared with the main algorithms in classic grammatical inference in terms of performance and accuracy. In addition, certain upgrades on the Elman RNN architecture are introduced to solve the following issues: (1) the generalization problem of dynamical systems when they are trained on a classification problem, (2) the over-regularization problem caused by L1 regularization, (3) to avoid quantization techniques of the internal state space, and (4) to bring out a new tool to analyze the behavior of the recurrent neural networks. These techniques provide meaningful interpretability: on one hand the neurons in the hidden layer are coding specific and interpretable patterns, and on the other hand the RNN seems to mimic the behavior of a finite state machine.

Keywords

Grammatical Inference, Recurrent Neural Networks, Deterministic Finite Automata

Índice

Listas

Lista de algoritmos

Lista de códigos

Lista de ecuaciones

Lista de figuras

Lista de tablas

1

Introducción

La inferencia de gramáticas consiste en aprender un lenguaje formal [1] a partir de un conjunto finito de datos, construyendo un modelo que reconozca aquellas características o patrones comunes, es decir, generar un conjunto de reglas, producciones o, si fuese el caso de una máquina de estados, transiciones que proporcionen la capacidad de reconocer los patrones de todo el conjunto de los datos. Un lenguaje formal [1] es un conjunto, que puede o no ser finito, de secuencias finitas de símbolos (cadenas) bien definidas por una estructura matemática que define su gramática. En los años 60, Noam Chomsky hizo una jerarquía de los distintos tipos de gramáticas formales [2] definiendo así los lenguajes recursivamente enumerables (tipo 0), los lenguajes dependientes del contexto (tipo 1), los lenguajes independientes del contexto (tipo 2) y los lenguajes regulares (tipo 3).

En la ciencia de la computación, la posibilidad de inferir la gramática de un lenguaje formal a partir de un conjunto finito de cadenas del lenguaje se ha convertido en un problema completamente matemático, donde el objetivo consiste en definir un algoritmo capaz de extraer las reglas correspondientes a la gramática del lenguaje formal a inferir. Con el avance del aprendizaje automático, concretamente las redes neuronales recurrentes (RNRs), se ha abierto un nuevo campo de investigación con un amplio abanico de posibilidades, entre las que se encuentra la extracción de reglas de una gramática a partir de una RNR entrenada con ejemplos del lenguaje. Esto cambia completamente el punto de vista de los algoritmos clásicos de inferencia gramatical.

Las RNRs son un tipo de redes neuronales artificiales especialmente diseñadas para modelar datos con una estructura temporal, ya que su arquitectura introduce una conexión en la que el estado interno de la red depende tanto de la entrada actual como del estado interno previo. Las RNRs se han aplicado en muchos ámbitos donde es necesario procesar datos secuenciales con una relación temporal, tales como el reconocimiento de voz [3,4], tratamiento del lenguaje natural [5,6] o composición de música [7], entre muchos otros.

En este trabajo se realiza un estudio de diferentes algoritmos clásicos de inferencia de gramáticas regulares y se desarrolla un modelo de RNR para abordar el mismo problema. En particular, los objetivos de este trabajo son los siguientes:

- Realizar un estudio de las principales alternativas de inferencia de gramáticas regulares, concretamente implementando los algoritmos clásicos $RPNI$ y $LSTAR$ (ver sección 2.2).
- Realizar un estudio del estado del arte de las técnicas de inferencia de gramáticas regulares y extracción de reglas utilizando RNRs (ver sección 2.3).
- Proponer ciertas mejoras en las RNRs con dos propósitos: mejorar la generalización de estos modelos cuando se enfrentan a un problema de clasificación de lenguajes regulares e introducir una metodología para mejorar la interpretabilidad de las RNRs (ver sección 3.3).
- Hacer un análisis comparativo entre las diferentes técnicas estudiadas en términos de rendimiento (ver sección 4.3).

Este documento está dividido en tres capítulos además de esta introducción: En el capítulo 2 se presenta el estado de la cuestión sobre el campo de investigación de inferencia de gramáticas regulares. En el capítulo 3 se describe el diseño de los experimentos y las mejoras propuestas a lo largo de este proyecto. Por último, antes de dar paso a las conclusiones finales, en el capítulo 4 se presentan las pruebas, evidencias, desarrollos y el análisis comparativo del proyecto. Adicionalmente se incluyen apéndices con el diseño y la implementación de los algoritmos necesarios. Concretamente, en el apéndice A se detalla el algoritmo de retropropagación, en el apéndice B se desglosan los algoritmos clásicos de inferencia de gramáticas regulares que se han implementado y, por último, en el apéndice C se proporciona la implementación de la RNR desarrollada.

El transcurso de este proyecto ha brindado la posibilidad de enviar dos contribuciones a conferencias internacionales [8, 9] con los resultados de esta investigación, concretamente aquellos relacionados con la RNR desarrollada en la sección 3.3 y las modificaciones introducidas para mejorar la interpretabilidad.

2

Estado del arte

Antes de comenzar con la cuestión que nos ocupa en este trabajo, es igualmente importante situar al lector en un contexto inicial, por un lado introduciendo los conceptos básicos a los que se hará referencia a lo largo de todo el documento y, por otro lado, proporcionando un estudio del estado de la cuestión sobre este campo de investigación. De esta manera, este capítulo se divide en los siguientes apartados: en la sección 2.1 se describen los conceptos que han sido considerados como básicos para situar al lector en un contexto adecuado. En la sección 2.2 se presentan algunos de los algoritmos clásicos de inferencia de gramáticas regulares teniendo en cuenta la naturaleza de los datos iniciales. Por último, en la sección 2.3 se presenta el estado del arte de la inferencia de gramáticas regulares utilizando las redes neuronales recurrentes.

2.1. Conceptos básicos

2.1.1. Lenguajes regulares: Expresiones Regulares y Autómatas Finitos Deterministas

Como precedente a la definición de un lenguaje regular, se introduce una gramática formal como una cuádrupla G = (N, Σ, P, A) donde N es un conjunto de símbolos no terminales, es decir, se pueden sustituir mediante una regla de producción (se suelen representar en mayúsculas); Σ es el alfabeto, símbolos finales que no pueden ser sustituidos (en este caso, en minúsculas); P es un conjunto de producciones o reglas de transformación; y A $\in$ N es el símbolo del axioma, es decir, el símbolo raíz. La diferencia entre los diferentes tipos de gramática que define Chomsky [2] se basa en las restricciones que tienen las producciones.

Los lenguajes regulares (LR) son el conjunto de los lenguajes formales más restrictivos de acuerdo a la jerarquía de Chomsky [2]. Se describen a partir de las gramáticas de tipo 3, también conocidas como gramáticas regulares, y son equivalentes a las expresiones regulares definidas por Stephen Kleene [10]. Estas expresiones están formadas por una secuencia de caracteres que constituyen el patrón de todas las cadenas que pertenecen a un determinado lenguaje (ver ejemplo en tabla 2.1, donde se

introduce el lenguaje que contiene todas las cadenas con un número par de símbolos a: $Paridad$). Las gramáticas regulares, que son aquellas que reconocen la clase de los lenguajes regulares, son las más restrictivas: el lado izquierdo de una producción debe contener un símbolo no terminal, mientras que el lado derecho puede contener símbolos terminales y no terminales siguiendo siempre la misma estructura de linealidad, es decir, los símbolos terminales aparecen siempre al inicio (gramática regular derecha) o siempre al final (gramática regular izquierda).

Gramática regular	**Expresión regular**
A $\rightarrow$ bA | aB | λ B $\rightarrow$ bB | aA	$b^*(ab^*ab^*)^*$

Tabla 2.1: Ejemplo de la descripción de un lenguaje regular con alfabeto Σ = {a, b} mediante su gramática regular derecha y su expresión regular equivalente: *cadenas con número par de símbolos* a. En la descripción de la gramática se están utilizando el símbolo | como operador booleano OR con el fin de agrupar aquellas producciones con el mismo símbolo no terminal a la izquierda y el símbolo λ como representación de la cadena vacía. En la expresión regular equivalente se sigue la nomenclatura de Kleene [10], donde el símbolo $*$ representa 0 o más repeticiones del símbolo o grupo al que aplica.

Por otro lado, es bien sabido que las cadenas pertenecientes a un lenguaje regular dado pueden ser identificadas mediante un autómata finito, es decir, existe una equivalencia directa entre una gramática regular dada, su expresión regular correspondiente y el autómata finito que la identifica [11]. Un autómata finito determinista (AFD), o máquina de estados determinista, es un modelo computacional abstracto formado por un número finito de estados internos y una función de transición que, a partir de una entrada, produce una determinada salida. Por definición, una máquina de estados es una quíntupla M = (Q, Σ, δ, q_0, F) donde Q es el conjunto de los estados internos del autómata, Σ es el alfabeto, δ: Q$\times\Sigma \rightarrow$ Q es la función de transición, $q_0 \in$ Q es el estado inicial y F $\subseteq$ Q es el conjunto de estados de aceptación (finales). El funcionamiento de los autómatas se basa en la función de transición δ, que dado un estado q_{actual} y un símbolo del alfabeto, devuelve un estado $q_{siguiente}$ al que se transita con dicho símbolo. Si después de procesar una cadena completa $w \in \Sigma^*$ el estado interno del autómata es un estado final $q_f \in$ F, entonces la cadena w pertenece al lenguaje, ya que cumple las producciones de la gramática formal. Además, se añade el concepto de determinismo cuando existe una única transición $\delta_{q,s} \in \delta$ para cada estado $q \in$ Q y símbolo $s \in \Sigma$. El conjunto de todas las cadenas aceptadas por el autómata es un lenguaje regular.

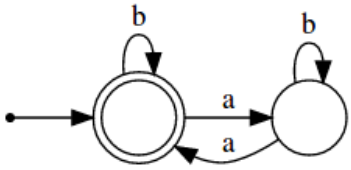


Figura 2.1: Diagrama de estados del AFD mínimo para el lenguaje regular $Paridad$.

En la figura 2.1 se muestra el diagrama de estados que representa el AFD correspondiente al problema $Paridad$. En este diagrama se define completamente la quíntupla del autómata: cada círculo identifica cada uno de los estados de Q; las flechas entre los estados y los símbolos sobre ellas representan las transiciones δ y el alfabeto Σ; el estado q_0 es el que tiene una flecha inicial; y los estados de aceptación F son aquellos con doble círculo.

2.1.2. Aprendizaje automático

El aprendizaje automático [12, 13] es un campo de la ciencia de la computación y la inteligencia artificial [14] que tiene como finalidad crear modelos capaces de diferenciar elementos de un conjunto de datos por inducción de acuerdo a sus características o atributos y agruparlos en clases de forma automática, es decir, sin la intervención humana. Cuando se utilizan estos modelos de clasificación se debe prestar especial atención a cómo se manipulan los datos, ya que no es buena práctica validar el modelo elegido utilizando los mismos datos con los que se ha entrenado. Por este motivo, es habitual dividir el conjunto de todos los datos en dos subconjuntos disjuntos: el conjunto de entrenamiento, con el que se entrena el modelo; y el conjunto de test, con el que se comprueba la eficiencia del modelo con datos no vistos previamente. El proceso del aprendizaje automático consta de cuatro fases diferenciadas:

- La recolección de los datos para analizarlos y clasificarlos.
- La intervención de un conocimiento experto externo que establece los atributos y el modelo a utilizar.
- El entrenamiento del modelo.
- La validación del modelo, de la que se pueden obtener dos resultados: o bien el entrenamiento es correcto y el modelo es el adecuado, o bien el resultado no es el adecuado y hay que modificar alguno de los puntos anteriores.

En general no hay un mejor modelo frente a otros, ya que depende del problema a resolver. Debe ser lo suficientemente complejo para capturar la información de los datos y lo suficientemente robusto para no ser sensible a un error de los datos.

2.1.3. Redes neuronales artificiales

Una red neuronal artificial es un modelo de aprendizaje automático basado en un paradigma inspirado en el funcionamiento biológico de las neuronas. M. Nielsen [15] describe así el arquetipo de este modelo: “In the conventional approach to programming, we tell the computer what to do [...] By contrast, in a neural network we don’t tell the computer how to solve our problem. Instead, it learns from observational data, figuring out its own solution to the problem”.

Si dos neuronas están conectadas sinápticamente y sus disparos están relacionados causalmente, se refuerza la conexión [16], intentando mantener cierto isomorfismo con la complejidad neurológica del modelo humano. La siguiente ecuación describe la activación de una neurona y con i conexiones sinápticas:

$$y = f(\sum_{0}^{i} w_i x_i + b) \tag{2.1}$$

donde x_i representa cada uno de los estímulos de entrada, w_i es el peso de la conexión entre x_i y la neurona de salida, b es el sesgo (bias) de activación de la neurona y f es una función de activación. Cabe destacar que, mientras que el argumento de la función f es completamente lineal, la función de activación introduce un alto grado de no-linealidad. Normalmente se puede aplicar una función escalón, la función sigmoidal o alguna función equivalente en forma, como la tangente hiperbólica.

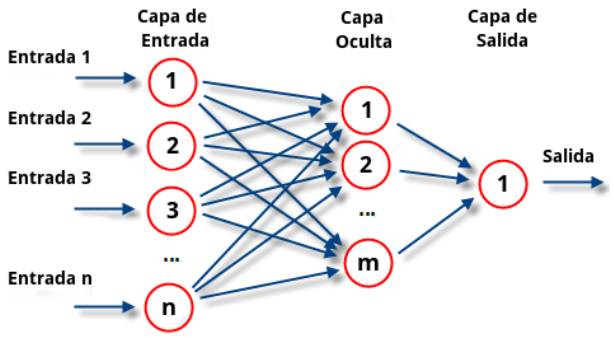


Figura 2.2: Red neuronal multicapa. Nótese que, aunque el diagrama tenga una única neurona en la capa de salida, es posible diseñar redes multicapa con las neuronas de salida necesarias. [Imagen creada por *Gengiskanhg* para Wikipedia - `https://commons.wikimedia.org/wiki/File:RedNeuronalArtificial.png`].

El entrenamiento de una red neuronal consiste, básicamente, en el cálculo de los valores de las conexiones (pesos) entre sus entidades (neuronas) para que, a partir de un conjunto de datos de entrada, se ajusten al resultado objetivo. Pese a que existen diferentes arquitecturas que definen las conexiones entre neuronas, la más utilizada cuando uno se enfrenta a un aprendizaje supervisado es la distribución por capas en una red con propagación hacia adelante (feedforward), donde cada capa está completamente conectada con la siguiente. Como se puede ver en la figura 2.2, las neuronas se agrupan en una capa de entrada $(input)$, una o varias capas ocultas $(hidden)$ y una capa de salida $(output)$ con la libertad de estar compuestas por un número diferente de neuronas en cada capa. Se puede observar la propagación de la información hacia adelante ya que la capa anterior solamente establece conexiones con la capa siguiente. Este modelo de red define la activación de las neuronas en la capa i siguiendo la siguiente ecuación:

$$z_i = f(W_i z_{i-1} + b_i) \tag{2.2}$$

donde z_i es el vector de salida de la capa i, W_i es la matriz de pesos que conecta la salida de la capa anterior (z_{i-1}) con la capa i, b_i es el vector de los bias para la capa i y f es la función de activación. Examinando esta simple ecuación, uno puede concluir que el cálculo de la activación de una capa completa consiste en aplicar una función altamente no-lineal a una sencilla operación matricial.

Descenso por gradiente: Retropropagación

El modelo de una red neuronal artificial multicapa requiere ajustar los parámetros hasta encontrar una buena aproximación de la solución del problema a resolver. Por tanto, es necesario un algoritmo que permita encontrar los pesos y bias para minimizar una función de coste que mida la discrepancia entre la salida de la red y el resultado objetivo durante una fase de entrenamiento. Típicamente se utiliza la función del error cuadrático medio cuando este modelo se enfrenta a un problema de regresión, mientras que en la resolución de problemas de clasificación se suele utilizar la función de entropía cruzada.

El objetivo del algoritmo de entrenamiento será, pues, minimizar esta función de coste utilizando la técnica del descenso por gradiente, normalmente añadiendo ciertas mejoras como un factor de aprendizaje adaptativo o momento, o utilizando métodos de segundo orden como Adam [17], Adagrad [18] o Adadelta [19]. Este algoritmo se conoce como *Retropropagación* (ver apéndice A), cuya idea principal consiste en seguir los siguientes pasos:

1.– Explorar hacia adelante calculando las salidas esperadas.

2.– Calcular el error mediante la función de coste.

3.– Volver hacia atrás para el cálculo de los gradientes.

4.– Devolver los pesos actualizados.

2.1.4. Redes neuronales recurrentes

Una red neuronal recurrente (RNR) es un tipo de red neuronal multicapa que presenta una relación de recurrencia en sus conexiones internas, consiguiendo que el estado interno de la red h_t dependa tanto de la entrada de la red x_t como de su estado anterior h_{t-1}. Se describe a continuación la arquitectura clásica de la red neuronal de Elman [20], cuyas funciones de activación se describen con las siguientes ecuaciones:

$$h_t = \sigma(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \tag{2.3a}$$

$$y_t = \sigma(W_{hy}h_t + b_y) \tag{2.3b}$$

donde x_t, h_t y y_t representan los vectores de activación de las capas de entrada, intermedia y de salida a tiempo t, respectivamente. El resto de los símbolos definen las matrices de pesos (W_{xh}, W_{hh} y W_{hy}), el vector de bias para cada capa (b_h y b_y), y la función de activación sigmoide (σ). Nótese la única diferencia en la ecuación 2.3a respecto a la activación de la red neuronal multicapa (ecuación 2.2), donde se introduce una recurrencia en la activación de la capa intermedia h_t, ya que esta depende del estado previo h_{t-1} controlado por la matriz de pesos W_{hh}.

Se muestra en la figura 2.3 un esquema ilustrativo de las conexiones de la RNR y el concepto de despliegue (unfold) habitual. Como se puede observar en la parte izquierda de la figura, esta arquitectura de red tiene una única capa oculta h donde existe una conexión recurrente consigo misma, representando el comportamiento de la ecuación 2.3a. Por otro lado se representa en la parte derecha de la figura el fenómeno conceptual conocido como $despliegue$, que permite comprender esta arquitectura como una red neuronal multicapa con un número a priori infinito de capas intermedias, donde cada capa se ve influenciada por la entrada de la secuencia temporal en el instante de tiempo anterior. Esta interpretación presenta un dilema en el cálculo del descenso por gradiente, ya que cada instante de tiempo depende de la activación de la capa oculta de la red en infinitos instantes anteriores. Es por este motivo por el que es indispensable definir un límite de despliegue seq_len con el que establecer una longitud máxima de dependencia temporal.

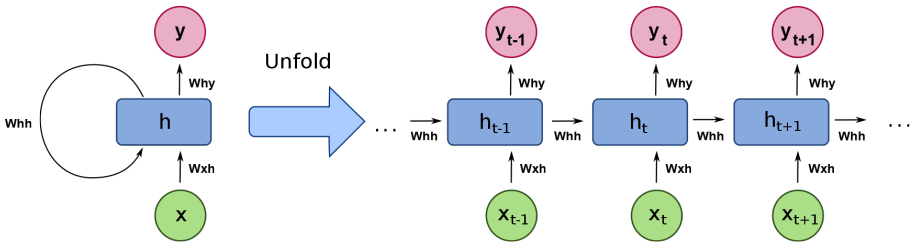


Figura 2.3: Ilustración del concepto de recurrencia en una RNR. [Imagen creada por *François Deloche* para Wikipedia - `https://commons.wikimedia.org/wiki/File:Recurrent_neural_network_unfold.svg`]

Aunque en este trabajo se ha decidido utilizar como base esta arquitectura, existen otros modelos de redes neuronales recurrentes con arquitecturas más complejas, como las RNR de segundo orden [21, 22], las LSTM [23] o las GRU [24].

2.2. Inferencia clásica de gramáticas regulares

Una vez se han definido los conceptos básicos, en esta sección se pretende realizar un resumen de los algoritmos clásicos de inferencia de gramáticas regulares más utilizados y, por tanto, estudiar la capacidad de generar un AFD a partir de una presentación de los datos S, basándose en [25]. Tal y como se describe en el libro, es necesario distinguir tres tipos de presentaciones de los datos S: (i) *texto*, cuando los datos incluyen exclusivamente cadenas aceptadas por el lenguaje a inferir (S_+) (ver sección 2.2.1); (ii) *informante*, cuando los datos incluyen cadenas aceptadas y rechazadas por el lenguaje, además de si lo son o no $(S_+ \cup S_-)$ (ver sección 2.2.2); (iii) *oráculo*, cuando existe un sistema experto externo al que hacer consultas para inferir el conocimiento (ver sección 2.2.3).

Sin embargo, antes de comenzar con el estudio de los diferentes planteamientos de inferencia de gramáticas regulares, es imprescindible definir el concepto de *identificación en el límite*. Por definición, una clase de lenguaje $\mathcal{C}(L)$ es identificable en el límite cuando existe un algoritmo que, a partir de un conjunto suficiente de cadenas de cualquier lenguaje de dicha clase, es capaz de dar la gramática correcta. En esta sección se argumenta si la clase de los lenguajes regulares $\mathcal{C}(LR)$ es identificable en el límite a partir de un texto, un informante o un oráculo, es decir, si existe algún algoritmo que pueda inferir los lenguajes regulares a partir de su presentación correspondiente.

2.2.1. Inferencia a partir de un texto

El planteamiento inicial de la inferencia de gramáticas regulares a partir de un texto (S_+) es la primera alternativa que, de forma lógica, cualquiera puede plantear: la posibilidad de aprender un lenguaje L a partir de un conjunto de cadenas positivas $S_+ \in L$. De la Higuera [25] describe perfectamente este primer planteamiento: “Learning from text consist on inferring from a presentation of examples that all come from the target language. The learner is asked to somehow generalize from the data it sees while not having counter-examples that would help it refrain from overgeneralising”.

A lo largo de este proyecto se han implementado algunos de los algoritmos de inferencia de gramáticas a partir de un texto descritos en [25] como tarea de aprendizaje y estudio. Sin embargo, aunque estos algoritmos son correctos para inferir la gramática para algunos subconjuntos de los LR (ver ejemplo de los lenguajes K-Testable en el apéndice B.1, que se describen como el subconjunto de los lenguajes regulares cuyas cadenas tienen la misma terminación con los k-1 últimos caracteres), se puede demostrar que $\mathcal{C}(LR)$ no es identificable en el límite a partir de un texto. Para comprender la demostración es necesario dar la siguiente definición: una clase de lenguaje es *superfinita* si contiene todos los posibles lenguajes finitos y, al menos, un lenguaje infinito. Por tanto, $\mathcal{C}(LR)$ es una clase superfinita.

Los AFD no son identificables en el límite a partir de un texto ya que, como son capaces de reconocer una clase de lenguajes superfinita, es imposible identificar la clase en el límite a partir de solamente casos positivos. Si el algoritmo converge en un lenguaje infinito L para un texto completo T en tiempo N, existe un lenguaje finito L' que contiene únicamente todas las cadenas en T para los que el algoritmo también converge en L, por lo que el algoritmo falla al identificar este lenguaje finito en el límite [26].

2.2.2. Inferencia a partir de un informante

Como se ha definido en la introducción de esta sección, se define un informante como la unión de dos conjuntos disjuntos: S_+, conjunto de cadenas aceptadas por el lenguaje; y S_-, conjunto de cadenas rechazadas por el lenguaje. En esta sección se presenta el algoritmo $RPNI$ y se demuestra que la clase de los lenguajes regulares es identificable en el límite a partir de un informante utilizando dicho algoritmo.

RPNI - Regular Positive and Negative Inference

Tal y como introduce [25]: "The idea is to greedily create clusters of states (by merging) in order to come up with a solution that is always consistent with the learning data". El algoritmo comienza con la generación de un autómata inicial a partir de las cadenas S del informante para, a partir de él, ir buscando posibles combinaciones de estados para conseguir autómatas equivalentes consistentes con la información dada. Para comprender el algoritmo completo es necesario definir dos conjuntos de estados que el algoritmo utiliza: (i) RED es el conjunto de estados que ya han sido procesados y no deben ser revisados de nuevo; y (ii) $BLUE$ es el conjunto de estados marcados como pendientes de revisión, que además tienen exactamente un único predecesor, es decir, existe una única transición $\delta_{q,s} = q_{BLUE}$ para cada $s \in \Sigma$.

El algoritmo $RPNI$ (apéndice B.2.1) consiste en, una vez se ha generado el autómata capaz de identificar las cadenas positivas, ir revisando cada uno de los estados $BLUE$ para comprobar si, después de combinarlo con algún estado RED, el autómata sigue siendo consistente con todos los datos, es decir, sigue rechazando todas las cadenas de S_-. Para definir correctamente el algoritmo, es necesario introducir las siguientes subrutinas: la primera función, *ConstruirPTA* (apéndice B.2.2), consiste en generar un autómata finito a partir del informante. Este autómata PTA se diferencia de un AFD por tener estados de *rechazo*, además de los habituales de *aceptación*. *Promocionar* (apéndice B.2.3) define la transformación de un estado $Q \in BLUE$ a RED. *Compatible* (apéndice B.2.4) comprueba si el autómata dado es consistente con el informante, es decir, sigue rechazando todas las cadenas de $S_- \in S$. *Fold* (apéndice B.2.5) es una función recursiva auxiliar que ejecuta el pliegue de las dos ramas a combinar. Por último, *Merge* (apéndice B.2.6) es la función que combina dos estados utilizando *Fold*.

Demostración: Identificación en el límite a partir de un informante

El fundamento del algoritmo $RPNI$ consiste en mantener la consistencia de los datos de la presentación hasta que, para todos los estados generados, se ha comprobado que el AFD no se puede reducir más sin dejar de ser consistente. Este algoritmo es capaz de generar un AFD que reconoce todas las cadenas del informante, es decir, aceptar las cadenas S_+ y rechazar las cadenas S_-. La demostración de que, efectivamente, la clase de los lenguajes regulares sí es identificable en el límite está implícita en la definición del propio algoritmo. En algún momento todas las cadenas de longitud $2n$-1, donde n es el número de estados del AFD objetivo, habrán aparecido en la presentación y habrán indicado que cualquier otro autómata consistente con los datos es mayor que el AFD objetivo. Por lo tanto, la clase de los lenguajes regulares es identificable en el límite a partir de un informante, ya que, a partir de cierto número de cadenas en la presentación, es indiferente añadir más cadenas ya que el mismo autómata objetivo será generado.

2.2.3. Inferencia a partir de un oráculo

El aprendizaje a partir de consultas es un paradigma que simula la existencia de un sistema experto adecuado al que realizar preguntas para obtener conocimiento. “A minimally adequate teacher is an oracle that can give answers to membership queries (MQ) and strong equivalence queries (EQ)” [25]. Una consulta MQ ejecuta una pregunta de clasificación al oráculo para obtener una respuesta binaria ($True$ o $False$), mientras que una consulta EQ ejecuta una pregunta al oráculo para obtener un *contraejemplo*, es decir, una cadena que no está siendo tratada correctamente, o, en caso contrario, la afirmación de equivalencia de conocimiento entre el oráculo y el sistema aprendiz.

LSTAR

El algoritmo $LSTAR$ es el que define este curioso paradigma de aprendizaje, apoyándose principalmente en la siguiente idea:

- Encontrar una tabla de observación, la cual representa el conocimiento del aprendiz.
- Preguntar con una consulta EQ si el conocimiento es equivalente al oráculo.
- En el caso de no serlo, actualizar el conocimiento con el contraejemplo proporcionado, ejecutando consultas MQ hasta alcanzar una nueva tabla de observación *cerrada*, *completa* y *consistente* y volver al paso anterior.
- Si lo es, el lenguaje ha sido aprendido.

por lo que, además de definir las subrutinas necesarias para poder dar el algoritmo $LSTAR$ completo (ver apéndice B.3.1), es necesario especificar la definición de una tabla de observación y sus condiciones de cierre, completitud y consistencia.

En primer lugar, una tabla de observación es una cuádrupla OT = (C, E, S, R), donde C es el conjunto color (RED o $BLUE$) al que pertenece el conocimiento, E es el conjunto de estados o cadenas tratadas, S es el conjunto de cadenas sucesoras y R es la respuesta del oráculo, que se puede entender como el conocimiento en sí. Se muestra en la tabla 2.2 un ejemplo de este tipo de tablas.

	λ	a
λ	0	1
a	1	0
b	1	0
aa	0	1
ab	1	0

Tabla 2.2: Ejemplo de tabla de observación. La primera columna representa el conjunto E, mientras que el resto incluye cada una de las cadenas del conjunto S. El color C se diferencia con la línea horizontal intermedia, donde la parte superior indica el conjunto RED y la inferior marca el conjunto $BLUE$. De esta forma, el conocimiento R se obtiene haciendo consultas MQ al oráculo con la cadena generada al concatenar el cada elemento de E con cada elemento de S. Nótese que para cada registro $e \in RED$ tienen que estar definidos otros registros en E (RED o $BLUE$) generados al concatenar e con cada símbolo del alfabeto del problema.

Por otro lado, una tabla de observación está: (i) *cerrada* cuando para cada uno de los registros en $BLUE$ existe un mismo conjunto de respuestas para cada elemento en S equivalente en algún registro de RED; (ii) *completa* cuando la tabla no tiene ningún registro sin la respuesta proporcionada por la consulta MQ correspondiente; y (iii) *consistente* cuando para cada pareja de registros RED equivalentes, siguen siendo equivalentes después de concatenarles cada elemento del conjunto S.

Entendidos todos los conceptos necesarios, se presentan ahora todas las subrutinas necesarias para completar correctamente el algoritmo $LSTAR$. El primer algoritmo, *CreaAutomata* (apéndice B.3.2), genera el AFD correspondiente a la tabla de observación dada, utilizando los registros RED como posibles estados del autómata. La siguiente subrutina, *Cerrar* (apéndice B.3.3), es la encargada de cerrar la tabla de observación dada, es decir, trasladar el registro $BLUE$ que provoca que la tabla no esté cerrada a RED, añadiendo todos los posibles sucesores a $BLUE$. La subrutina *Inicializar* (apéndice B.3.4), es la encargada de generar la tabla de observación inicial, generando en RED el registro λ y en $BLUE$ todos los posibles sucesores a partir del alfabeto (si Σ = {a, b}, se generan en $BLUE$ los registros a y b). La subrutina *Consistente* (apéndice B.3.5) trata de hacer consistente la tabla de observación añadiendo un nuevo sufijo al conjunto S, concretamente aquel sufijo que provoca que la tabla sea inconsistente entre dos registros. Por último, la subrutina *UsarEQ* (B.3.6) actualiza la tabla de observación a partir del contraejemplo dado por la consulta EQ, añadiendo a RED todos los posibles segmentos de la respuesta y a $BLUE$ todos sus posibles sucesores, en el caso en el que no existan previamente dichos registros.

Demostración: Identificación en el límite a partir de un oráculo

El algoritmo $LSTAR$ finaliza correctamente generando el AFD completo con n estados. Queda claro que cualquier AFD consistente con la tabla de observación tiene al menos tantos estados como registros diferentes hay en RED. Además, si la tabla está cerrada, la construcción del autómata es única, por lo que la tabla solo puede crecer de forma vertical hasta n distintos registros. "If the algorithm has built a table with n obviously different rows in RED, and n is the size of the minimal DFA for the target, then it is the target. The algorithm therefore is correct and terminates" [25].

La demostración de que este algoritmo identifica en el límite la clase de los lenguajes regulares a partir de un oráculo es equivalente a la demostración de la identificación en el límite a partir de un informante. El algoritmo se basa en mantener la consistencia con los datos de entrenamiento del oráculo, de forma que ambos sean capaces de generar el mismo conocimiento. La tabla de observación se genera de forma que se reproduzca el total de los datos, por lo que, para cualquier lenguaje regular aprendido por el oráculo, el algoritmo $LSTAR$ habrá aprendido el AFD correspondiente. Si el oráculo identifica en el límite la clase de los lenguajes regulares, el aprendiz lo identifica de igual manera, demostrando la inferencia en el límite a partir de un oráculo.

2.3. Inferencia mediante RNR

Cualquier problema computable puede ser resuelto por una máquina de Turing y es posible demostrar que una RNR es un modelo Turing-completo [27], por lo que una RNR debe ser capaz de computar un lenguaje regular. Además, su comportamiento debería ser equivalente a un AFD, tal y como argumenta [28]. Desde el planteamiento de esta idea en los años 90 [29], muchos autores han estudiado el paralelismo entre ambos modelos y la habilidad de las RNR de aprender lenguajes formales.

Siguiendo la revisión de H. Jacobsson realizada en 2005 [30], existe una abrumadora cantidad de artículos en los que se busca esta equivalencia, entre los que se encuentra no solo la extracción de reglas para transformar una RNR a un AFD, sino también otras técnicas como *clusterización* [20, 31], que, a groso modo, consiste en el procesado de los datos, tanto durante el entrenamiento como cuando este finalice, para agrupar un conjunto de objetos con características similares, o la proyección del espacio de estados interno [32, 33], normalmente utilizando algoritmos de reducción de dimensionalidad. Es interesante mencionar también una revisión reciente de Q. Wang [34], donde expone sus resultados obtenidos en esta materia utilizando RNRs de segundo orden y hace referencia a las *Tomita Grammars* [35] para ilustrarlos, y el trabajo realizado por J. J. Michalenko [36] donde utiliza técnicas similares a nuestro planteamiento de proyección a un espacio de dos dimensiones.

El objetivo principal de la extracción de reglas gramaticales para inferir lenguajes regulares utilizando redes neuronales recurrentes típicamente ha consistido en aplicar diversas técnicas de cuantización, que tratan de transformar el espacio interno de estados de la red en un conjunto discreto de estados que corresponden a los estados del AFD que se quiere extraer. Sin embargo, este hito es un punto de conflicto en el campo de investigación de inferencia de gramáticas utilizando modelos dinámicos, ya que, tal y como indica [37], el mero hecho de transformar un sistema continuo en discreto supone enfrentarse a un riesgo que puede provocar errores en la ejecución, ya que supone a priori una incoherencia. Tal y como comenta Jacobsson en su revisión [30] y si se realiza una comparativa con la inferencia clásica de gramáticas regulares, aunque los métodos de inferencia de las RNRs son capaces de generar un AFD equivalente al lenguaje que se pretende inferir, se ha demostrado que estas redes presentan un problema de generalización con cadenas que se diferencian drásticamente del conjunto de entrenamiento de la red (ver ejemplo en la sección 4.1.1, figura 4.3), por lo que [37] gana fuerza en su argumentación, proponiendo la no identificación en el límite de las RNRs sobre la clase de los lenguajes regulares. Sin embargo, en este ámbito de aprendizaje, ya que estas redes son capaces de aprender los casos habituales y solo presentan problemas con cadenas poco frecuentes, son una herramienta aceptada por la comunidad científica como método de extracción de reglas del lenguaje. Es más, en la mayoría de los casos las reglas extraídas generalizan mejor que las propias RNRs [34].

Aunque existen muchas variantes y nuevas alternativas de inferencia de AFD a partir de métodos de clustering u otros algoritmos, no existe actualmente una interpretación que pueda demostrar que las RNRs estén implementando directamente un AFD, ya que es un modelo considerado como una caja negra [30]. Sin embargo, tal y como se ha demostrado en las publicaciones realizadas en el transcurso de este proyecto [8, 9], introduciendo una cantidad controlada de ruido en la función de activación y aplicando regularización L1, es factible obtener de un modelo dinámico, tal y como son las RNRs, una respuesta estable, discreta y fácilmente interpretable sin necesidad de aplicar un mecanismo de discretización, contrario a como se consideraba previamente (ver sección 4.1.5). El proyecto en general, y [8, 9] en particular, se han basado en términos de interpretabilidad en un trabajo reciente de A. Karpathy [38], en el que, utilizando redes LSTM, no solo demuestra la eficiencia de generación de lenguaje natural, sino que introduce una simple pero interesante técnica de análisis del comportamiento interno de la red, utilizando un mapa degradado de colores. Además, demuestra que ciertas neuronas tienen una funcionalidad concreta en la resolución del problema.

3

Diseño

De acuerdo a los objetivos de este proyecto, después de haber desarrollado el estado del arte de la inferencia de gramáticas regulares utilizando tanto los algoritmos clásicos (sección 2.2) como las técnicas de extracción de reglas utilizando RNRs (sección 2.3), corresponde desarrollar el diseño de los experimentos y las mejoras propuestas sobre la arquitectura de la red de Elman inicial descrita en la sección 2.1.4.

Entrando más en detalle, en un primer apartado (3.1) se describe tanto el método de generación de los datos como los lenguajes regulares que se van a utilizar a lo largo de los experimentos y resultados. En el apartado 3.2 se van a determinar los algoritmos clásicos que se van a emplear para realizar las pruebas, siguiendo el criterio de la identificación en el límite. En el siguiente apartado (3.3) se introduce el diseño de la red neuronal recurrente, que, como se ha comentado previamente, es una modificación de la red de Elman descrita en la subsección 2.1.4 para dar una respuesta a las mejoras propuestas a nivel de generalización e interpretabilidad. Por último, en el apartado 3.4 se describen los experimentos ejecutados para realizar una comparativa de resultados en términos de eficiencia.

3.1. Descripción y generación de los datos

En este proyecto se han considerado los lenguajes regulares sobre el alfabeto {a, b} definidos en la tabla 3.1, que incluye los siguientes problemas: dos problemas sencillos con los que se inició el proyecto ($Paridad$ y BxA) y las bien conocidas *Tomita grammars* [35] que son consideradas un estándar en la investigación sobre inferencia de gramáticas regulares.

Debido al diseño de la RNR para tratar las cadenas del lenguaje como una serie de datos con una estructura temporal, donde en cada instante de tiempo se procesa un símbolo de la cadena, se ha decidido incluir un símbolo de escape $ para identificar la separación de cada cadena, por lo que se puede interpretar como el inicio de cadena. La introducción de este símbolo permite que la red sea capaz de procesar el conjunto de cadenas sin interrupción, lo cual facilita la implementación de este tipo de problemas de clasificación.

Para la generación de los conjuntos de entrenamiento y test se ha implementado un generador de cadenas aleatorias con los símbolos del alfabeto y las probabilidades de cada símbolo (ver tabla 3.2), mientras que la aceptación o rechazo de cada instante de tiempo con la cadena se ha generado utilizando directamente los autómatas que definen los lenguajes regulares.

Nombre	**Lenguaje regular**	**Expresión Regular**
$Paridad$	Cadenas con paridad de a's.	b*(ab*ab*)*
BxA	Cadenas que empiezan por b y terminan por a.	b(a+b)*a
$Tomita1$	Cadenas con solo a's.	a*
$Tomita2$	Cadenas con solo secuencias de ab.	$(ab)^*$
$Tomita3$	Cadenas sin un número impar de a's seguidas por un número impar de b's.	b*[aa(aa)*b*+a(aa)*bb(bb)*]* (a+λ)
$Tomita4$	Cadenas con un número menor que 3 b's consecutivas.	$(a+ba+bba)^*(bb+b+\lambda)$
$Tomita5$	Cadenas de longitud par con número par de a's	[aa+bb+(ab+ba)(aa + bb)* (ab+ba)]*
$Tomita6$	Cadenas con diferencia de a's y b's múltiplo de tres.	[ba+(a+bb)(ab)*(b+aa)]*
$Tomita7$	Cadenas con la forma $X_1Y_1X_2Y_2$ donde $X_{1,2}$ son dos subcadenas con solo b's e $Y_{1,2}$ son dos subcadenas con solo a's.	b*a*b*a*

Tabla 3.1: Descripción de los lenguajes regulares utilizados en este trabajo.

La tabla 3.2 describe los conjuntos para entrenar y validar la RNR. Para ello se han generado cinco datasets que serán empleados durante todos los experimentos: un conjunto de entrenamiento ($train$) que contiene 50.000 símbolos con la misma probabilidad de aparición de los símbolos a y b, y varios conjuntos de test con diferentes características: el conjunto big se ha generado con las mismas condiciones que el conjunto de entrenamiento, exceptuando su tamaño (100.000 símbolos). El conjunto $long$ contiene únicamente 20.000 símbolos y tiene una menor probabilidad de aparición del símbolo \$, consiguiendo como resultado un conjunto de cadenas con una longitud notoriamente mayor. Por último, los conjuntos $all\ as$ y $all\ bs$ representan el mismo concepto pero con símbolos contrarios: ambos tienen 15.000 símbolos y una muy elevada probabilidad de aparición del símbolo a o b, respectivamente. El objetivo de estos dos últimos datasets es, precisamente, mostrar empíricamente el efecto del problema de la generalización de estos modelos.

Todos estos datasets se han generado de la siguiente manera: (i) un fichero de entrada con una secuencia aleatoria con los símbolos a, b y \$, y (ii) un fichero de salida con la correspondiente respuesta del AFD para cada instante de tiempo con cada símbolo.

Data	# chars	a prob.	b prob.	\$ prob.	avg len	min len	max len
train	50000	0.45	0.45	0.1	8.9	0	95
big	100000	0.45	0.45	0.1	9.0	0	81
long	20000	0.495	0.495	0.01	88.3	0	477
all a*s*	15000	0.98	0.01	0.01	113.5	0	566
all b*s*	15000	0.01	0.98	0.01	95.8	0	475

Tabla 3.2: Descripción de los conjuntos de $train$ y $test$ para entrenar y validar la RNR. La tabla muestra el número de caracteres, la probabilidad de distribución de cada símbolo y, como ejemplo ilustrativo, máximo, mínimo y media de la longitud de las cadenas generadas para el problema $Paridad$.

3.2. Algoritmos clásicos

La inferencia de gramáticas clásica utilizando los algoritmos presentados en la sección 2.2 proporciona dos técnicas con la capacidad de identificar en el límite la clase de los lenguajes regulares: la inferencia de gramáticas a partir de un informante utilizando el algoritmo $RPNI$ y la inferencia a partir de un oráculo utilizando el algoritmo $LSTAR$. Ambos algoritmos han sido completamente implementados en lenguaje $Python$ siguiendo el pseudo-código de los apéndices B.2.1 y B.3.1 con los propósitos de analizar su eficiencia frente al modelo de la red neuronal y proporcionar una técnica de extracción del AFD equivalente de la red.

3.3. Descripción de la RNR

A lo largo de este proyecto se ha planteado utilizar diferentes herramientas de $Python$, tales como *Tensorflow* [39] o *Keras* [40], que son potentes librerías enfocadas al diseño de modelos de aprendizaje profundo y redes neuronales recurrentes. Además, se ha probado también (utilizando *Tensorflow*) una arquitectura de red más compleja y, a priori, más competente: las redes LSTM [23]. Sin embargo, el propio curso del proyecto ha requerido un rediseño hacia una arquitectura más sencilla para tener mayor control, ya que las librerías mencionadas no son tan flexibles a la hora de modificar su implementación interna. En este trabajo se ha desarrollado una RNR tomando como base la arquitectura de la red de Elman descrita en la sección 2.1.4 utilizando el software proporcionado por A. Karpathy [41], a partir de la cual se ha añadido cierta funcionalidad para proponer principalmente dos mejoras frente al problema de la inferencia de gramáticas regulares.

En primer lugar, como se introduce en la sección 2.3 y como se muestra en la sección de resultados (4.1.1), las RNRs presentan un problema de generalización cuando se enfrentan al problema de la identificación en el límite de los lenguajes regulares. Este problema sucede, por ejemplo, al introducir en el conjunto de validación cadenas que la red no ha visto durante el entrenamiento, y es especialmente relevante cuando se validan cadenas compuestas por mayoritariamente un único símbolo (el resultado obtenido por los conjuntos $all\ as$ y $all\ bs$ muestra empíricamente esta condición). Se plantea entonces una modificación de la red en su función de activación de la capa oculta para, además de utilizar la tangente hiperbólica, añadir en la fase de entrenamiento ruido Gausiano en la recurrencia, tal y como se muestra en la siguiente ecuación:

$$h_t = \tanh(W_{xh}x_t + (W_{hh} + X_\nu \mathbb{I})h_{t-1} + b_h) \tag{3.1}$$

donde X_ν es una variable aleatoria con una distribución normal centrada en 0 y varianza ν e $\mathbb{I}$ es la matriz Identidad $n_h \times n_h$. Nótese que, de esta manera, se aplica el ruido de manera proporcional a la activación de cada neurona para que no entre en conflicto con la regularización L1 (ver sección 4.1.4). Con esta modificación se resuelve el problema de la generalización.

En segundo lugar, las RNRs siguen siendo consideradas como una caja negra capaz de clasificar, predecir o generar información de forma muy eficiente, pero sin proporcionar ninguna interpretación significativa más allá de una combinación de pesos que hace eficaz la ejecución. Para mejorar la interpretabilidad se propone un entrenamiento de la red minimizando la función de coste de entropía cruzada con regularización L1, ya que esta técnica reduce de forma agresiva los pesos de la red, consiguiendo discriminar aquellas neuronas que no tienen una funcionalidad clara, siguiendo la siguiente ecuación:

$$L = -\sum_{t=1}^{n}[\hat{y}_t \log(y_t) + (1-\hat{y}_t)\log(1-y_t)] + \gamma(||W_{xh}||_1 + ||W_{hh}||_1 + ||W_{hy}||_1) \tag{3.2}$$

donde $\hat{y}_t$ es la salida esperada, γ es el factor de regularización, el parámetro n es el número de pasos de despliegue (seq_len) y la expresión $||.||_1$ representa la norma L1. Con esta segunda modificación se pretende reducir el número de neuronas a analizar, consiguiendo así que la interpretabilidad quede reducida a entender el comportamiento de aquellas neuronas activas en la capa oculta. Sin embargo, la aplicación de la regularización L1 provoca, en alguno de los casos, un nuevo problema de sobre-regularización, donde la RNR no es capaz de converger a la solución óptima. Para evitar este problema se introduce el concepto de $shock$, que consiste en una reactivación de las neuronas de la RNR cuando no es capaz de aprender debido a esta sobre-regularización, siguiendo la siguiente ecuación:

$$W_{xy} = W_{xy} + X_\zeta \tag{3.3}$$

donde W_{xy} es la matriz de pesos correspondiente y X_ζ es una variable aleatoria con una distribución normal centrada en 0 con varianza ζ. El shock se aplica cuando el coste no ha disminuido en C iteraciones. Cada vez que el shock es activado, el límite de iteraciones de entrenamiento de la red aumenta en *limite_inicial*$*2^{-n}$ iteraciones, siendo n el número de shocks aplicados. Con esta última modificación se espera que la red sea capaz de resolver el problema de la sobre-regularización, consiguiendo una red que supere el problema de la generalización al mismo tiempo que es fácilmente interpretable.

Como se verá en la sección de resultados (4.1.6), la introducción de estas mejoras aporta, asimismo, un efecto de cuantización automática del espacio de estados interno de la red, consiguiendo así un modelo pseudo-discreto que parece resolver uno de los principales hitos de discrepancia en la inferencia de gramáticas mediante redes neuronales: aquel que consiste en la incoherencia matemática de obtener un sistema discreto, como lo son los AFD, a partir de un sistema continuo y dinámico, como lo son las RNRs, ya que en la mayoría de variantes que se han estudiado se realiza una cuantización artificial del espacio de estados interno de la red con el objetivo de discretizar el sistema, provocando posibles errores en el momento en que se ajustan los umbrales de decisión.

3.3.1. Interpretabilidad

Como propuesta para proporcionar interpretabilidad en la RNR, se ha decidido realizar un mapa degradado de colores para representar la aceptación o rechazo de la cadena en cada instante de tiempo t (dorado = *Aceptar*, blanco = *Rechazar*) y la activación de cada neurona de la capa oculta, desde cian (-1), pasando por blanco (0) hasta rojo (+1). Utilizando esta metodología se puede observar en el comportamiento de la red el reconocimiento de ciertos patrones, que se pueden entender como las instrucciones que debe seguir para aprender la gramática correspondiente.

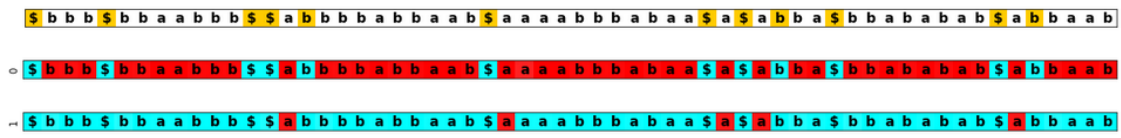


Figura 3.1: Ejemplo ilustrativo de la interpretabilidad de una red utilizando el problema $Tomita2$.

De esta manera, se muestra en la figura 3.1 un ejemplo de este mapa de colores correspondiente a la activación de la red con el problema $Tomita2$ para ilustrar el método de interpretación: en la primera fila se muestra la decisión de la red en cada instante de tiempo, marcando en dorado la aceptación. Nótese que, tal y como se ha descrito en la sección 3.1, el símbolo $ marca el inicio de cadena. A partir de la segunda fila, cada una de ellas se corresponde a la activación de cada una de las neuronas de la capa oculta de la red.

En el ejemplo la red tiene $n_h = 2$ neuronas en la capa oculta y se puede ver un claro comportamiento discreto de las activaciones de las neuronas, ya que, aunque se esté mapeando un degradado de colores, solamente se pueden apreciar los tonos de saturación (rojo y cian). A lo largo del capítulo de resultados se utilizan estos mapas para interpretar el funcionamiento de las redes.

3.4. Descripción de los experimentos de la comparativa

Para cada problema se compararán los algoritmos $RPNI$ y $LSTAR$ con la configuración de la RNR que proporciona un 100 % de acierto en todos los datasets. La comparación y análisis de resultados se ha planteado siguiendo los hitos presentados a continuación: (1) un análisis de rendimiento en el que se compara el tiempo promedio de ejecución de 20 iteraciones para cada problema y algoritmo, donde se presenta la disyuntiva entre un entrenamiento prolongado y eficiente en términos de interpretabilidad, debido a la presencia de la regularización L1 y el shock, frente a un entrenamiento con buen rendimiento, donde se prioriza el tiempo de ejecución, y (2) un análisis de rendimiento de ambos algoritmos clásicos con la RNR, donde las dos alternativas anteriores juegan un papel relevante.

4

Desarrollo y resultados

Una vez descritos los objetivos, el estado del arte y el diseño de este proyecto, la finalidad de este último capítulo consiste en presentar el desarrollo y los resultados obtenidos utilizando las herramientas descritas en el capítulo anterior. Se han realizado pruebas con diferentes configuraciones de la RNR desarrollada (apéndice C) al mismo tiempo que se analiza la interpretabilidad de la misma, finalizando con un método de análisis de su comportamiento interno, mostrando que puede estar implementando un autómata (sección 4.1). Se han probado los algoritmos clásicos, demostrando empíricamente la identificación en el límite de los lenguajes regulares y brindando un método de extracción del AFD a partir de la RNR (sección 4.2) y, por último, se ha realizado una comparativa de los algoritmos clásicos frente a la RNR en términos de rendimiento (sección 4.3).

4.1. Inferencia mediante RNR e interpretabilidad

Utilizando la RNR detallada en la sección 3.3 se plantean las siguientes alternativas de configuración, marcadas por el trascurso del proyecto: en la sección 4.1.1 se presentan los resultados obtenidos al ejecutar la red de Elman inicial (ecuaciones 2.3a y 2.3b), analizando los primeros comportamientos en términos de interpretabilidad y descubriendo de forma empírica el problema de la generalización. En la sección 4.1.2 se presenta la primera mejora, donde se introduce la regularización L1 como discriminador de neuronas, proporcionando una interpretabilidad bastante significativa para algunos problemas. En la sección 4.1.3 se muestran los resultados obtenidos con la segunda propuesta, presentando una inyección controlada de ruido en la función de activación de la capa oculta de la red como solución al problema de generalización. En la sección 4.1.4 se combinan ambas mejoras y se analizan los resultados obtenidos, descubriendo ahora el problema de la sobre-regularización. A continuación, en la sección 4.1.5 se introduce la última mejora, el shock, consiguiendo el objetivo esperado: la red identifica, al menos, los nueve lenguajes regulares que se plantean en este proyecto. En la última sección (4.1.6) se desarrolla una técnica de análisis del comportamiento interno de la red para demostrar que está actuando de forma equivalente al AFD que corresponde. La implementación en $Python$ desarrollada durante este proyecto se puede revisar en el apéndice C.

Para cada experimento realizado se han entrenado 20 redes diferentes con una inicialización aleatoria de las matrices de pesos en el rango [-0.01, 0.01]. La red se ha entrenado fijando el límite inicial en 100.000 iteraciones (50 épocas), alcanzando un máximo de 300.000 iteraciones para los casos en que corresponda según el número de aplicaciones del $shock$ (ecuación 3.3). La función de coste se minimiza utilizando el descenso por gradiente estándar, fijando el factor de aprendizaje en 0.01 y un $unfold$ de 25. Todos los resultados que se muestran en las siguientes subsecciones son el promedio de estos 20 entrenamientos para cada conjunto de parámetros iniciales (ruido ν, regularización γ, shock ζ). Durante todos los experimentos se va a mostrar el mapa de colores definido en la sección 3.3.1 para aquellos problemas que se han considerado más ilustrativos [1].

4.1.1. Red de Elman inicial

La primera configuración de parámetros de la red consiste en no aplicar ninguna modificación respecto a la red original de Elman, lo que implica fijar los parámetros de ruido ν, regularización L1 γ y shock ζ a 0.

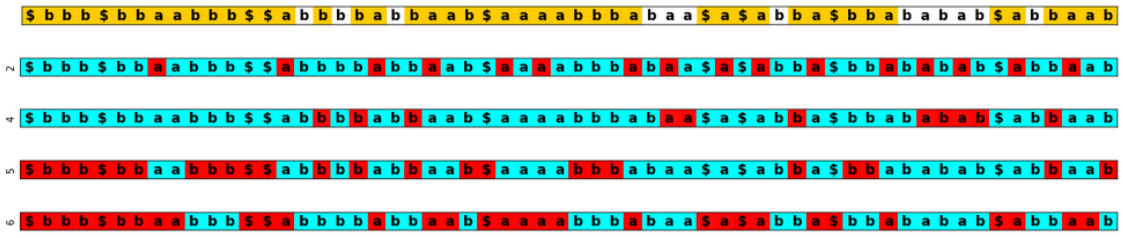


Figura 4.1: Mapa de colores para el problema $Paridad$ ($\nu = 0{,}0$, $\gamma = 0{,}0$, $\zeta = 0{,}0$) con n_h = 4.

Pese a tener un correcto funcionamiento en algunos casos, esta configuración presenta los dos problemas ya mencionados. En primer lugar, en la figura 4.1 se muestra una ejecución del problema $Paridad$ con un 0 % de error en los 5 datasets con la configuración mencionada. En ella se puede observar que la RNR está utilizando todos los recursos disponibles para codificar la solución, por lo que la interpretabilidad se dificulta a la hora de entender qué está haciendo cada neurona. Este mismo resultado se puede observar en el resto de problemas planteados en la tabla 3.1 y con un número mayor de n_h.

Si se ajusta el número de neuronas se obtiene una mayor interpretabilidad (ver figura 4.2, donde se muestra un ejemplo con 0 % de error en los cinco datasets y 2 neuronas), ya que cada neurona define una funcionalidad concreta: la primera neurona (N0) tiene una activación positiva (rojo) ante el símbolo a en posición par y el símbolo $ cuando la cadena debe ser rechazada, mientras que la segunda neurona (N1) se activa (rojo) con los símbolos de aceptación, es decir, las b's cuando hay un número

[1] Los resultados para el conjunto completo de las *Tomita Grammars*, incluyendo los mapas de colores, la proyección Isomap del espacio de estados interno y el autómata extraído se han publicado para [9] y están disponibles en nuestro repositorio de GitHub: `https://github.com/slyder095/coliva_llago_icann2019`

Figura 4.2: Mapa de colores para el problema $Paridad$ ($\nu = 0{,}0$, $\gamma = 0{,}0$, $\zeta = 0{,}0$) con $n_h = 2$.

par de a's en la cadena y el símbolo $ cuando la cadena previa es aceptada. La principal observación que se extrae, viendo este comportamiento, es el planteamiento de una mayor interpretabilidad cuanto menor sea el número de neuronas activas. Este es el origen de introducir la regularización L1, ya que aportará una reducción dinámica de neuronas activas (ver subsección 4.1.2).

En segundo lugar, para algunos problemas con cierta complejidad, como son BxA, $Tomita3$, $Tomita5$ o $Tomita7$, existe el problema del error en la generalización de la red. En la figura 4.3 se muestra un ejemplo del problema para el lenguaje BxA. Esta es la principal causa de la introducción de ruido en la función de activación de la capa oculta durante la fase de entrenamiento (ver subsección 4.1.3).

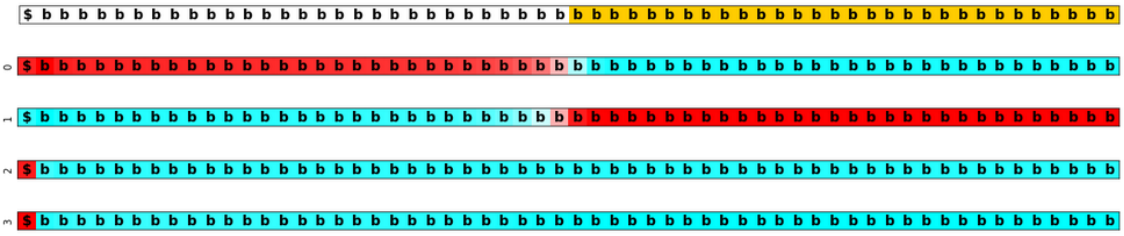


Figura 4.3: Mapa de colores para el problema BxA ($\nu = 0{,}0$, $\gamma = 0{,}0$, $\zeta = 0{,}0$) con $n_h = 4$ mostrando el problema de la generalización. La transición de colores para el mismo símbolo cuando debe mantenerse en el mismo estado provoca un comportamiento erróneo en la red, pasando de blanco ($Rechazar$) a dorado ($Aceptar$).

4.1.2. Regularización L1

La introducción de regularización L1 permite a la red ser capaz de reducir el número de neuronas activas de forma dinámica, "apagando" aquellas neuronas que no son estrictamente necesarias para resolver el problema. De esta forma, la red aumenta considerablemente la posibilidad de ser interpretada con mayor facilidad.

En la figura 4.4 se muestra un ejemplo de la eficacia de la regularización L1 con el problema $Paridad$. Como era de esperar, la red ha reducido el número de neuronas al mínimo necesario para aprender el lenguaje (ver efecto causado en las neuronas N0 y N1), permitiendo estudiar nuevamente la interpretabilidad de la red: la primera neurona (N2) reacciona con una activación positiva con las a's pares y el símbolo $ cuando la cadena previa debe ser rechazada, mientras que la segunda neurona

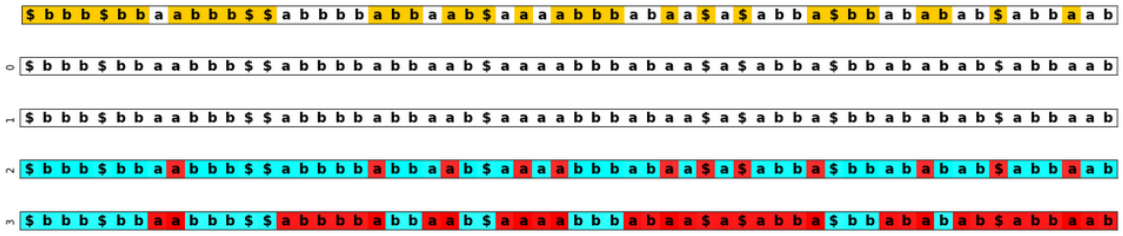


Figura 4.4: Mapa de colores para el problema $Paridad$ ($\nu = 0$, $\gamma = 10^{-4}$, $\zeta = 0$) mostrando el efecto de la regularización L1 con n_h = 4. Nótese que solamente 2 neuronas están activas, habiendo reducido los pesos de las dos neuronas restantes aproximadamente a 0.

(N3) se activa en negativo con las b's cuando las a's son pares y el símbolo $ cuando la cadena previa debe ser aceptada. Como conclusión, la regularización L1 proporciona a la red la interpretabilidad esperada a costa de tener activas menos neuronas.

4.1.3. Ruido en la función de activación

La introducción de una moderada cantidad de ruido en la función de activación de la capa oculta de la RNR (ver ecuación 3.1) permite mejorar considerablemente el resultado obtenido al enfrentarla contra los conjuntos de test "extremos" ($all\ as$ y $all\ bs$), donde se podía observar el problema de la generalización. Como ilustración a este problema y su resolución, se muestran en la figura 4.5 cien ejecuciones distintas de la red para dos configuraciones diferentes.

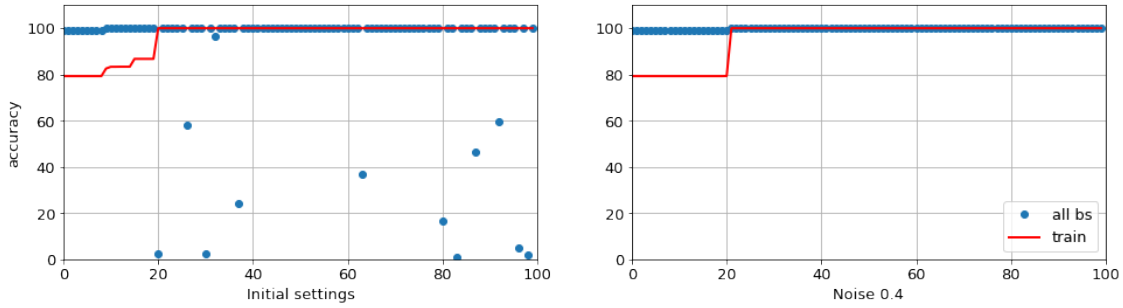


Figura 4.5: Se muestra el porcentaje de acierto del problema BxA frente al identificador de la ejecución para los conjuntos de $train$ y $allbs$ ordenados por el porcentaje de acierto de $train$ de menor a mayor. A la izquierda se muestra el resultado de 100 pruebas con la configuración inicial ($\nu = 0$, $\gamma = 0$, $\zeta = 0$). A la derecha se muestra el resultado de 100 pruebas añadiendo ruido ($\nu = 0{,}4$, $\gamma = 0$, $\zeta = 0$).

Como se puede observar en ambos casos, un 20 % de las ejecuciones no es capaz de converger a un 100 % de acierto en el conjunto de entrenamiento, por lo que en ese tramo no aplica analizar el problema de la generalización. En el 80 % restante la principal observación es que, mientras que sin ruido hay ciertas ejecuciones que tienen un claro problema de generalización (puntos que destacan por debajo del porcentaje de $train$), introduciendo ruido la red es capaz de generalizar correctamente en

todos los casos. En conclusión, haber introducido ruido hace que la red tenga una fase de entrenamiento más compleja, pero en caso de converger a la solución esperada, la RNR habrá resuelto el problema de la generalización frente a nuevas cadenas que no ha visto durante la fase de entrenamiento.

4.1.4. Combinación: ruido y regularización

Combinar ruido y regularización L1 permite a la red generalizar correctamente y ser más interpretable. En un principio, unir ambos factores parecería tener un efecto contrario, ya que el ruido activaría aquellas neuronas que se han regularizado. Sin embargo, con la arquitectura de la red descrita en la sección 3.3, el ruido se inyecta de forma proporcional a la activación de la propia neurona, por lo que ignorará aquellas neuronas regularizadas.

Figura 4.6: Mapa de colores para el problema BxA ($\nu = 0{,}2$, $\gamma = 10^{-4}$, $\zeta = 0$) mostrando un resultado con 0 % de error en los cuatro datasets de $test$ con n_h = 6.

En la figura 4.6 se muestra un ejemplo de ejecución frente al problema BxA, donde se ha resuelto el problema de generalización y se ha reducido el número de neuronas a las necesarias por la red, proporcionando además una buena interpretación: La primera neurona (N0) se activa en negativo (cian) cuando la cadena debe ser aceptada, es decir, cuando aparece una a en una cadena que ha empezado con b. La segunda neurona (N1) se activa en negativo (cian) cuando aparece una b en una cadena que ha empezado con b. La última neurona (N3) se activa en positivo (rojo) cuando aparece el símbolo $\$$. Este resultado es, en este caso concreto, directamente equivalente a su AFD, donde cada neurona se comporta como un estado del autómata correspondiente.

Un resultado similar se puede observar en la figura 4.7, donde se muestra un ejemplo de ejecución para el problema $Tomita3$: La primera neurona (N0) se activa en negativo (cian) cuando la cadena debe ser rechazada, es decir, cuando entra en el estado de $Error$. La segunda neurona (N1) se activa en positivo (rojo) con las b's pares detrás de una a impar. La tercera neurona (N3) se activa en negativo (cian) con todas las a's impares después de cualquier secuencia de b's. La última neurona (N4) se activa en positivo (rojo) con todas las b's pares después de una a impar. Con una interpretación similar a esta, es fácil observar como algunas de las neuronas (N1 y N3) están funcionando como neuronas auxiliares que ayudan a las otras neuronas (N0 y N4) a codificar correctamente la solución esperada.

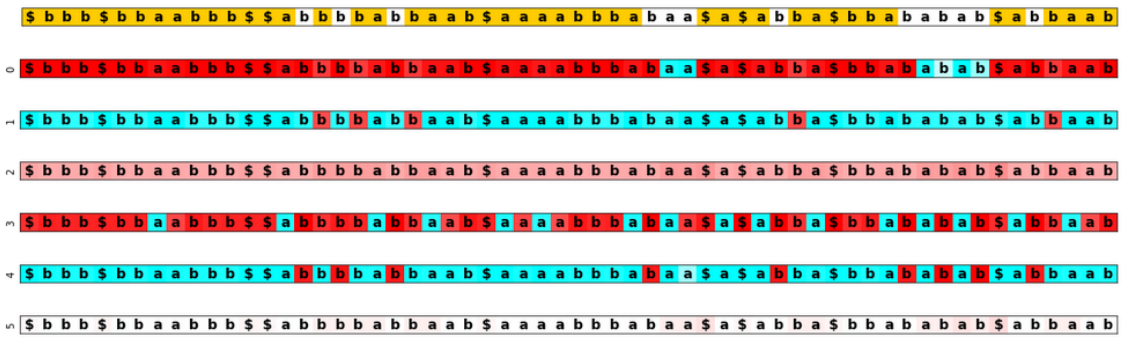


Figura 4.7: Mapa de colores para el problema $Tomita3$ ($\nu = 0{,}4$, $\gamma = 10^{-4}$, $\zeta = 0$) mostrando un resultado con 0 % de error en los cinco datasets con n_h = 6. Nótese que la neurona 2, aunque está siempre ligeramente activa, se puede ignorar en términos de interpretabilidad, ya que no tiene ningún comportamiento representativo.

Sin embargo, como se puede observar en la figura 4.8, que es un ejemplo de ejecución del problema $Tomita5$, puede aparecer el problema de la sobre-regularización, que se produce cuando la regularización L1 silencia más neuronas de las que la red necesita para converger a un buen resultado, obteniendo un fallo en la clasificación. Como se puede observar, la red está utilizando una neurona (N1) para reconocer el símbolo de inicio $ y una neurona (N4) como contador de paridad de la longitud de cada cadena. Con estos dos comportamientos la red está clasificando como $Aceptar$ aquellas cadenas de longitud par, dejando atrás una condición necesaria para cumplir el problema planteado por la gramática $Tomita5$. Sería necesaria una neurona más para comprobar la paridad de alguno de los símbolos restantes (a o b), ya que sería suficiente para clasificar correctamente todas las cadenas, como se muestra en la figura (4.10).

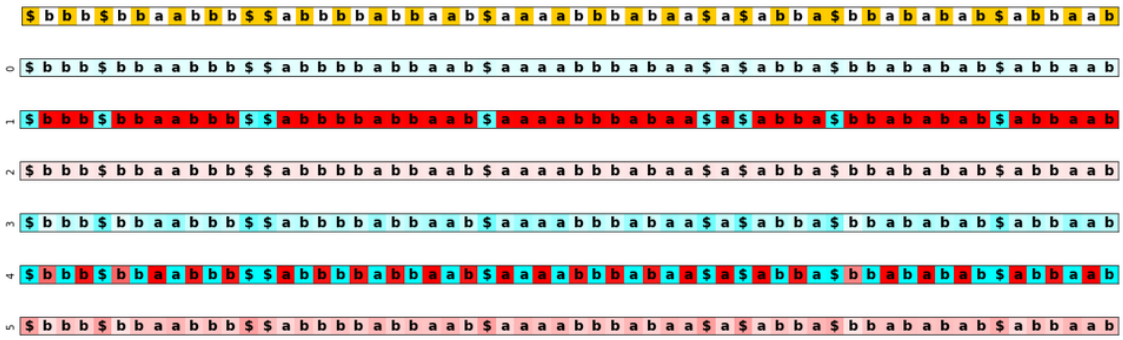


Figura 4.8: Mapa de colores para el problema $Tomita5$ ($\nu = 0{,}6$, $\gamma = 10^{-4}$, $\zeta = 0$) mostrando un resultado con el problema de sobre-regularización. Nótese que la regularización está anulando las neuronas 0, 2, 3 y 5.

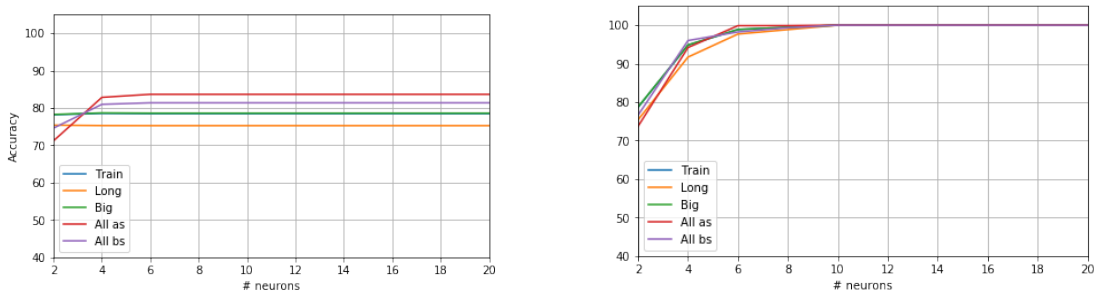
Como resumen, se muestra en la tabla 4.1 el resultado de la media de 20 ejecuciones diferentes para cada problema con la configuración ($\nu = 0{,}6$, $\gamma = 10^{-4}$, $\zeta = 0$), donde se pueden observar dos comportamientos: (i) se obtiene un porcentaje de acierto muy elevado en todos los datasets, y (ii) el entrenamiento no converge ($Tomita5$ y $Tomita6$) debido a la sobre-regularización.

20 n_h	*Paridad*	*BxA*	*T1*	*T2*	*T3*	*T4*	*T5*	*T6*	*T7*
train	100.0	100.0	100.0	100.0	100.0	100.0	78.47±0.0	72.27±0.0	100.0
big	100.0	100.0	100.0	100.0	100.0	100.0	78.61±0.0	72.02±0.0	100.0
long	100.0	100.0	100.0	100.0	100.0	100.0	75.26±0.0	67.24±0.0	100.0
all as	100.0	100.0	100.0	100.0	100.0	100.0	83.65±0.0	66.32±0.0	99.95±0.24
all bs	100.0	100.0	100.0	100.0	100.0	100.0	81.37±2.77	69.23±2.79	98.26±5.31

Tabla 4.1: Resultados en promedio de 20 ejecuciones con la configuración de ruido y regularización ($\nu = 0{,}6$, $\gamma = 10^{-4}$, $\zeta = 0$) con n_h = 20 para todos los problemas planteados en este documento.

4.1.5. Añadiendo shock: alcanzando el máximo

Con el fin de solucionar el problema de la sobre-regularización, se añade a la red el mecanismo de $shock$ descrito en la sección 3.3. Este permite a la red reactivar las neuronas regularizadas cuando no es capaz de aprender durante cierto número de épocas (5.000 en todos los experimentos realizados). Como ejemplo de los resultados de esta mejora, se muestra en la figura 4.9 el problema $Tomita5$, cuyo resultado con shock mejora considerablemente con un número suficiente de neuronas.



(a) $Tomita5$ sin shock

(b) $Tomita5$ con shock

Figura 4.9: Porcentaje de acierto en promedio frente al número de neuronas de la capa oculta para el problema $Tomita5$. A la izquierda se muestra la configuración sin shock ($\nu = 0{,}6$, $\gamma = 10^{-4}$, $\zeta = 0$). A la derecha se muestra la configuración con shock ($\nu = 0{,}6$, $\gamma = 10^{-4}$, $\zeta = 0{,}5$).

Como se puede observar, mientras que en la gráfica de la izquierda (sin shock) el porcentaje de acierto se estanca en torno a un 80 %, provocando el efecto mostrado en la figura 4.8 donde la red está sobre-regularizada, en la gráfica de la derecha (con shock) el porcentaje de acierto alcanza un 100 % en los cinco datasets con un número suficiente de neuronas. Se muestra en la figura 4.10 un ejemplo de ejecución para el problema $Tomita5$ con n_h = 8 aplicando ruido, regularización L1 y shock. En la figura se puede observar la resolución del problema que se planteaba con el mismo lenguaje en la figura 4.8, añadiendo una neurona auxiliar para completar los requisitos del problema. Con esta combinación es posible reconocer el lenguaje $Tomita5$ y proporcionar un 100 % de acierto.

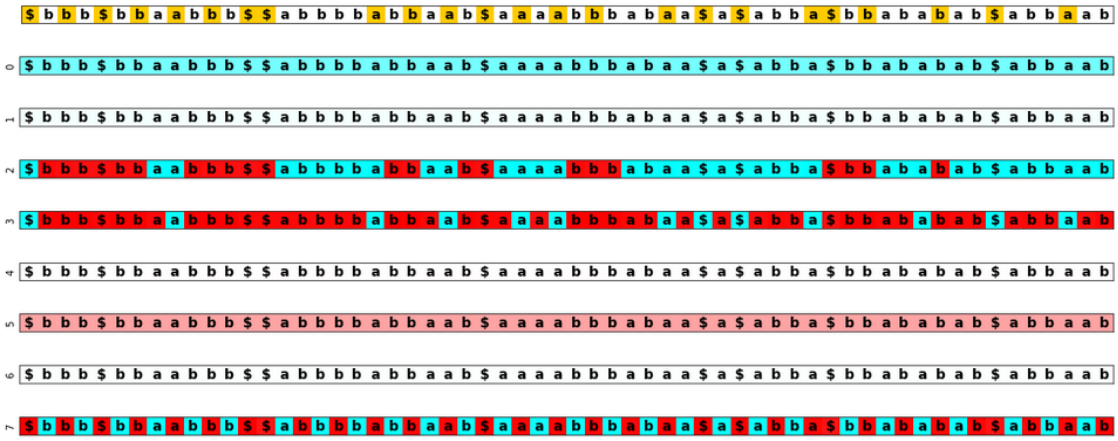


Figura 4.10: Mapa de colores para el problema $Tomita5$ ($\nu = 0{,}6$, $\gamma = 10^{-4}$, $\zeta = 0{,}5$) con n_h = 8 mostrando un resultado con porcentaje de error 0 % para los cinco datasets. Nótese que las neuronas 0 y 5 no aportan ninguna información significativa.

En la tabla 4.2 se puede observar cómo, para todos los lenguajes regulares planteados en este documento, la RNR con ruido, regularización L1 y shock es capaz de obtener un 100 % de acierto para todos los datasets generados.

20 n$_h$	*Paridad*	*BxA*	*T1*	*T2*	*T3*	*T4*	*T5*	*T6*	*T7*
train	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
big	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
long	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
all as	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
all bs	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0

Tabla 4.2: Resultados en promedio de 20 ejecuciones con la configuración de ruido, regularización y shock ($\nu = 0{,}6$, $\gamma = 10^{-4}$, $\zeta = 0{,}5$) con n$_h$ = 20 para todos los problemas planteados en este documento.

4.1.6. Análisis del comportamiento interno de la red

Se ha desarrollado una red que resuelve los nueve problemas planteados y, además, proporciona una interpretabilidad bastante significativa a partir de las herramientas descritas en la sección 3.3.1. Además, en esta última subsección se muestra que, al menos para los lenguajes regulares utilizados, la red desarrollada discretiza automáticamente el espacio de estados interno del sistema sin necesidad de aplicar ninguna cuantización, definiendo ciertos patrones que la red tiene en cuenta para proporcionar la salida esperada. El análisis de las transiciones en el espacio de estados interno de la red demuestra que aparentemente está implementando internamente el AFD equivalente, por lo que puede ser utilizada para extraer el lenguaje regular correspondiente.

Isomap

El método de análisis del comportamiento interno de la red consiste en realizar una proyección a dos dimensiones mediante el algoritmo *Isomap* [42] para analizar la activación de la capa oculta y extraer las transiciones que se definen en el espacio de estados interno de la red. La red desarrollada discretiza el espacio de estados interno, agrupándose de forma automática en un conjunto finito de estados [2]. La metodología seguida consiste en: (1) generar una cadena con la misma distribución de símbolos que el conjunto $train$ con una longitud fija (en todos los casos es 1.000) que contenga exclusivamente cadenas aceptadas (el símbolo previo al $ se predice como $aceptar$) para extraer la salida de la capa oculta y proyectarla a 2 dimensiones, y (2) obtener las transiciones de forma directa, ya que en el instante t la red transita al *macroestado* x_{t+1} con el símbolo de entrada. En la figura 4.11 se muestra la proyección Isomap de un ejemplo de ejecución para el problema $Tomita3$.

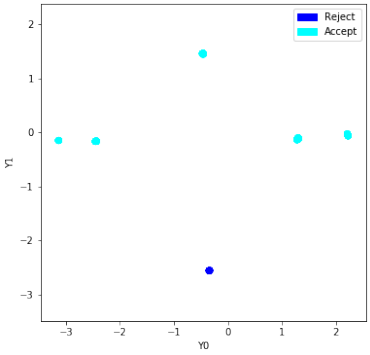


Figura 4.11: Proyección Isomap de la activación del espacio de estados interno de la red para el problema $Tomita3$. Los estados de aceptación se proyectan en cian, mientras que los de rechazo se proyectan en azul. Nótese que se están proyectando 1.000 instantes de tiempo consecutivos pero todos han sido discretizados por la red, formando seis macroestados.

En ella se puede observar cómo la red está agrupando su espacio de estados interno en seis *macroestados* bien definidos. Por tanto, solamente es necesario obtener las transiciones siguiendo la metodología descrita anteriormente. En la tabla 4.3 se muestran las proyecciones de las transiciones, donde en cada fila se fija como cluster inicial el sombreado con un cuadrado, mientras que en cada columna se dibuja en color rojo el estado destino.

Por ejemplo, la primera fila contiene el estado denominado A (arriba-centro) sombreado en gris. Desde este estado, la red transita al estado E con el símbolo a, se queda en el estado A con el símbolo b y transita a los estados A y D con el símbolo $. Sobre esta tabla de transiciones se pueden hacer dos observaciones principalmente: (i) las transiciones parecen ser deterministas exceptuando el símbolo de escape $, que parece comportarse con un no determinismo; (ii) la matriz de transiciones no es completa ya que solo se utiliza un conjunto de estados internos que conducen siempre a la aceptación de la cadena, por lo que aquellas transiciones que conducen a un estado “trampa” de rechazo no

[2] Jacobsson [30] denomina como macroestado la agrupación de diferentes estados de la red en la proyección a dos dimensiones.

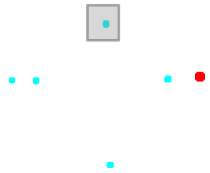
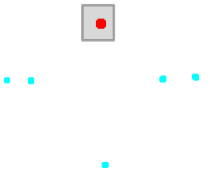
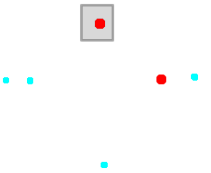
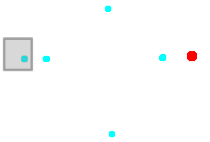
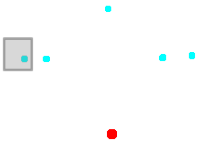
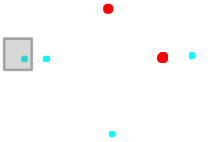
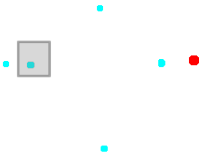
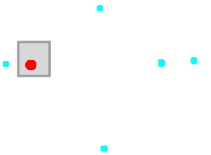
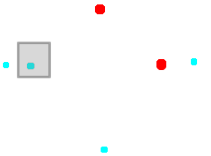
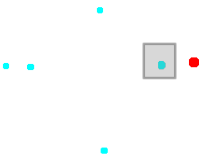
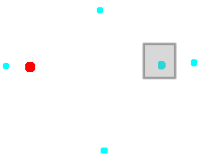
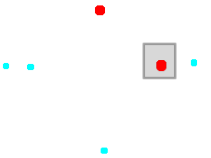
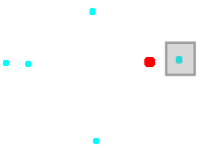
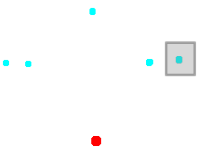
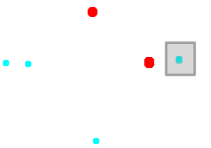
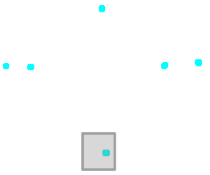
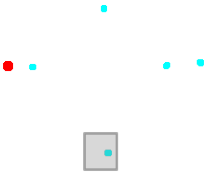
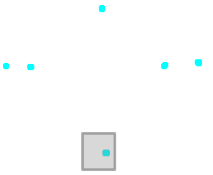
Nombre	Transiciones con *a*	Transiciones con *b*	Transiciones con $
A			
B			
C			
D			
E			
F			

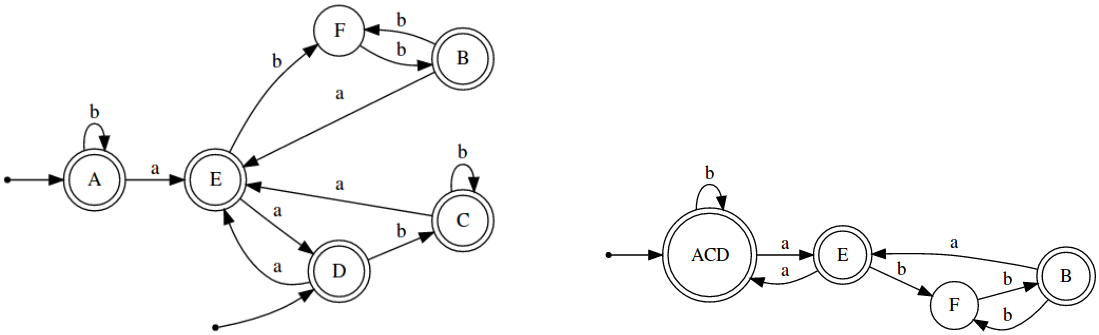
Tabla 4.3: Proyección de las transiciones entre estados en el espacio de estados interno de la red para el problema $Tomita3$. Cada fila muestra cada uno de los estados, mientras que cada columna muestra con qué símbolo se está transitando.

Symbol	state A	state B	state C	state D	state E	state F
a	E	E	E	E	D	-
b	A	F	C	C	F	B
$	A,D	A,D	A,D	A,D	A,D	-

Tabla 4.4: Tabla de transiciones obtenidas a partir de la tabla 4.3 para el problema $Tomita3$.

aparecen en la proyección. Estas dos observaciones son válidas para todas las ejecuciones realizadas. Se muestra en la tabla 4.4 un resumen de las transiciones obtenidas mediante las proyecciones mostradas en la tabla 4.3.

Sin embargo, la observación de que las transiciones con el símbolo de escape no son deterministas parece no tener relevancia. Este inusual comportamiento podría explicarse al observar que el símbolo $ debe reiniciar la red a su condición inicial ignorando completamente lo que haya sucedido previamente. Es coherente pensar que este complejo comportamiento es difícil de gestionar por la red, por lo que la aparición de varios estados iniciales podría ayudarla a tratar con todas las posibilidades. A pesar de esta observación, los estados iniciales forman un único estado siempre que se minimice el autómata obtenido, como se puede observar en la figura 4.12. Este resultado se ha comprobado con todas las pruebas realizadas con los diferentes problemas considerados.



(a) AFD extraído inicial

(b) AFD minimizado

Figura 4.12: AFD resultante de la extracción mediante el algoritmo Isomap para el problema $Tomita3$ descrito en la tabla 4.3 (a) y el autómata mínimo equivalente (b).

4.2. Inferencia mediante algoritmos clásicos

Tal y como se menciona en la sección 3.2, aunque se han implementado todos los algoritmos descritos a lo largo de este trabajo (ver sección 2.2), los dos algoritmos que se tratan en este apartado son $RPNI$ y $LSTAR$ debido a la identificación en el límite de la clase de los lenguajes regulares a partir de un informante o un oráculo. Para cada experimento se ejecuta cada algoritmo una única vez, ya que la ejecución no debe cambiar si el conjunto de datos de entrenamiento es igual, y se comprueba la eficacia del algoritmo directamente con el AFD extraído en cada caso.

4.2.1. RPNI

Como se ha descrito en la sección 2.2.2, el algoritmo $RPNI$ identifica en el límite la clase de los lenguajes regulares. Se han ejecutado diferentes configuraciones de los datos de la presentación, y se puede confirmar que con un número suficiente de cadenas el algoritmo es capaz de inferir el autómata finito determinista correspondiente. Durante las pruebas ejecutadas se han podido extraer dos conclusiones: (1) el número de cadenas tiene una importancia relevante ya que es necesario un número suficiente de estados intermedios para que el algoritmo sea capaz de converger a la solución correcta, y (2) es posiblemente más significativa la longitud máxima de las cadenas de entrenamiento, ya que existe un límite inferior con el que el algoritmo no es capaz de inferir el autómata independientemente del número de cadenas.

Siguiendo estas dos conclusiones, se muestra en la tabla 4.5 el valor de longitud máxima de las cadenas que al menos debe tener el conjunto de datos de entrenamiento del algoritmo para que sea capaz de inferir correctamente el AFD correspondiente a cada lenguaje regular.

Paridad	BxA	Tomita1	Tomita2	Tomita3	Tomita4	Tomita5	Tomita6	Tomita7
3	4	1	2	5	3	4	3	5

Tabla 4.5: Longitud mínima necesaria para que el algoritmo $RPNI$ infiera correctamente el AFD para todos los problemas descritos en la sección 3.1.

Por otro lado, se muestran en la figura 4.13 dos ejemplos de ejecución del algoritmo para el problema BxA con diferente longitud máxima de cadena de entrenamiento. Como se puede observar, independientemente del número de cadenas aceptadas y rechazadas, la longitud máxima de las cadenas o el número de estados del PTA que se está generando, el número de pasos es exactamente el mismo en ambas ejecuciones, y ambos finalizan con el mismo resultado. Este comportamiento se puede observar en todas las ejecuciones de todos los problemas a los que se ha enfrentado este algoritmo. Sin embargo, aunque se hará un análisis más detallado del rendimiento de este algoritmo en la sección 4.3, cabe destacar el crecimiento exponencial del número de estados respecto a la longitud máxima de la cadena como punto clave de rechazo de este algoritmo como alternativa viable.

```
> Problema: BxA
> Longitud máxima de cadena: 8
> Número de cadenas aceptadas: 88
> Número de cadenas rechazadas: 272

> Comienza el algoritmo RPNI...
  > Generando el PTA inicial...
  > Obteniendo el alfabeto...

--------------------
PASO 1
BLUE: 1
RED: 1
NUM ESTADOS: 145
--------------------
PASO 2
BLUE: 2
RED: 2
NUM ESTADOS: 145
--------------------
PASO 3
BLUE: 1
RED: 2
NUM ESTADOS: 75
--------------------
PASO 4
BLUE: 2
RED: 3
NUM ESTADOS: 75
--------------------
PASO 5
BLUE: 1
RED: 3
NUM ESTADOS: 41
--------------------
PASO 6
BLUE: 0
RED: 3
NUM ESTADOS: 3
------- FIN -------
```

(a) Ejecución con longitud máxima 8

```
> Problema: BxA
> Longitud máxima de cadena: 14
> Número de cadenas aceptadas: 286
> Número de cadenas rechazadas: 915

> Comienza el algoritmo RPNI...
  > Generando el PTA inicial...
  > Obteniendo el alfabeto...

--------------------
PASO 1
BLUE: 1
RED: 1
NUM ESTADOS: 841
--------------------
PASO 2
BLUE: 2
RED: 2
NUM ESTADOS: 841
--------------------
PASO 3
BLUE: 1
RED: 2
NUM ESTADOS: 524
--------------------
PASO 4
BLUE: 2
RED: 3
NUM ESTADOS: 524
--------------------
PASO 5
BLUE: 1
RED: 3
NUM ESTADOS: 299
--------------------
PASO 6
BLUE: 0
RED: 3
NUM ESTADOS: 3
------- FIN -------
```

(b) Ejecución con longitud máxima 14

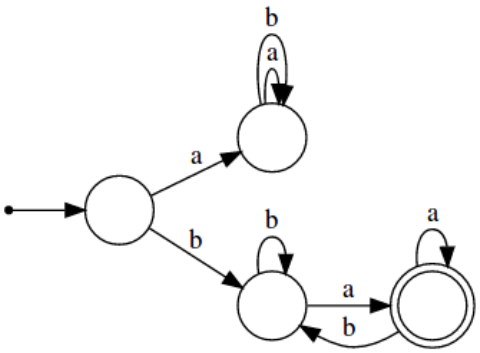
Figura 4.13: Ejemplo de ejecución del algoritmo $RPNI$ para el problema BxA.

4.2.2. LSTAR

El algoritmo $LSTAR$ es la segunda alternativa de inferencia de gramáticas regulares clásica, en este caso utilizando un sistema experto externo como oráculo. Tal y como se menciona en la demostración de la identificación en el límite de la inferencia de gramáticas a partir de un oráculo, si este sistema experto es capaz de identificar en el límite la clase de los lenguajes regulares, el algoritmo funcionará correctamente e inferirá el AFD mínimo. Como se demuestra en [28] y se ha demostrado empíricamente a lo largo de este proyecto, las RNRs son capaces de computar cualquier lenguaje regular, por lo que se pueden aplicar como oráculo para el algoritmo $LSTAR$, de forma que no sea necesario someterla a una herramienta de proyección dimensional que puede provocar pérdida de información significativa en caso de querer extraer el AFD correspondiente. Esta idea se basa en el trabajo realizado por G. Weiss [43], donde se utiliza una modificación del algoritmo para extraer el autómata al mismo tiempo que la red converge.

```
> Comienza el algoritmo LSTAR...
  > Mientras la respuesta EQ no sea OK:
    > Debe ser cerrada y consistente...
      > Cerrada y consistente
    > Consulta EQ...
      > Contraejemplo: ba
    > Debe ser cerrada y consistente...
      > Cerrada
      > No consistente...
        > Adding column: a
      > Cerrada y consistente
    > Consulta EQ...
      > Contraejemplo: aba
    > Debe ser cerrada y consistente...
      > Cerrada
      > No consistente...
        > Adding column: ba
      > Cerrada y consistente
    > Consulta EQ...
> OK
```

(a) Ejemplo de ejecución del problema BxA



(b) AFD inferido por el algoritmo

Figura 4.14: Ejemplo de ejecución del algoritmo $LSTAR$ para el problema BxA utilizando como oráculo la RNR desarrollada con un 100 % de acierto en los cinco datasets.

En la figura 4.14 se muestra un ejemplo de ejecución del algoritmo aprendiendo el problema BxA utilizando como oráculo la red entrenada con ruido, regularización y shock con un 100 % de acierto en los cinco datasets utilizados. Es interesante observar que el AFD extraído es completo, ya que incluye el estado “trampa”.

4.3. Comparación y análisis

En esta última sección se va a realizar un análisis comparativo de los distintos algoritmos en términos de rendimiento y se va a presentar la disyuntiva entre un entrenamiento de la red prolongado y eficiente en términos de interpretabilidad frente a un entrenamiento más rápido buscando exclusivamente la eficacia de la red. Para todos los resultados mostrados a continuación se ha realizado un promedio de 20 ejecuciones diferentes.

Eficacia e interpretabilidad

Tanto la red desarrollada como los algoritmos de inferencia de gramática implementados no presentan ninguna complicación en términos de eficacia. Por un lado, la RNR es capaz de aprender cada uno de los lenguajes y obtener un 100 % de acierto en todos los test. Por otro lado, se ha demostrado que los algoritmos $RPNI$ y $LSTAR$ identifican en el límite la clase de los lenguajes regulares y extraen directamente el AFD correspondiente: el algoritmo $RPNI$ es capaz de generar el autómata mínimo siempre y cuando haya un conjunto suficiente de datos en la presentación, y el algoritmo $LSTAR$ tiene la única complicación de necesitar un sistema externo para generar el AFD.

Sin embargo, en términos de interpretabilidad de la red no se debe pecar de pretencioso, ya que, aunque en la mayoría de pruebas realizadas se obtenga una interpretación bastante significativa del comportamiento de las neuronas, los resultados obtenidos no siempre son fácilmente interpretables más allá de observar la activación de ciertas neuronas cuando ha sucedido un evento concreto en el tiempo anterior.

Rendimiento

Respecto al rendimiento de los algoritmos y la red neuronal, se ha tenido en cuenta tanto el tiempo promedio de ejecución como la memoria necesaria. Se muestra en la tabla 4.6 el promedio de tiempo de ejecución de cada problema utilizando las siguientes configuraciones del algoritmo $RPNI$ y la RNN: (1) la diferencia entre *RPNI-6* y *RPNI-14* es la longitud máxima de las cadenas de la presentación, mientras que (2) las dos configuraciones de red (*RNN-validación* y *RNN-límite*) se diferencian en la condición de parada de la red. Por un lado, *RNN-validación* tiene como condición de parada superar una validación cada 10.000 iteraciones: alcanzar un 100 % de acierto durante el entrenamiento. Por otro lado, la condición de parada de *RNN-límite* es exclusivamente alcanzar el límite establecido (inicialmente 100.000). Cabe destacar que esta condición está ligada al número de shocks que reciba la red, tal y como se describe en la sección 3.3.

	RPNI-6	**RPNI-14**	**LSTAR**	**RNN-validación**	**RNN-límite**
Paridad	0.16±0.02	54.78±0.54	0.06±0.003	23.78±0.21	219.61±1.56
BxA	0.07±0.002	19.35±0.09	0.12±0.002	23.90±0.30	271.56±94.30
Tomita1	0.01±0.02	0.01±0.00	0.07±0.01	23.83±0.21	217.70±0.53
Tomita2	0.01±0.00	0.02±0.01	0.09±0.01	23.83±0.23	219.01±2.09
Tomita3	0.12±0.02	14.18±0.13	0.13±0.002	24.12±0.29	221.02±1.38
Tomita4	0.15±0.02	23.92±0.24	0.11±0.001	24.15±0.30	220.88±1.22
Tomita5	0.39±0.01	54.94±0.35	0.10±0.002	90.34±41.96	453.72±97.48
Tomita6	0.24±0.02	60.85±0.17	0.08±0.003	80.62±11.23	423.29±0.78
Tomita7	0.15±0.002	4.22±0.02	0.12±0.003	26.40±7.62	212.12±0.38

Tabla 4.6: Tiempo de ejecución en promedio para dos longitudes máximas de cadenas de entrenamiento en $RPNI$, el algoritmo $LSTAR$ y dos entrenamientos de la red neuronal, uno con condición de parada con validación y otro sin condición de parada.

De la tabla se extraen las siguientes conclusiones: (i) el tiempo de ejecución del algoritmo $RPNI$ aumenta de forma exponencial con la longitud máxima de las cadenas del conjunto de entrenamiento tal y como se muestra en la sección 4.2.1 (ver también figura 4.15). Este comportamiento es lógico ya que el algoritmo genera un PTA inicial con a lo sumo 2^L-1 estados, siendo L la longitud máxima de una cadena. (ii) El algoritmo $LSTAR$, aunque a priori sea el más eficiente en términos de rendimiento, por sí solo no parece el más apropiado para inferir una gramática a partir de un conjunto de datos, ya que

puede no proporcionar información suficiente o, en el peor de los casos, introducir datos erróneos en el sistema por causas externas (este mismo argumento también es válido para el algoritmo $RPNI$). Sin embargo, tal y como se muestra en la sección 4.1.6, es un eficiente e interesante método de extracción del AFD a partir de una RNR. (iii) Las RNRs parecen ser un arma de doble filo, ya que aumentar el rendimiento conlleva perder la capacidad de ser fácilmente interpretable en la mayoría de los casos. Esto se debe a la regularización L1, ya que, de forma intrínseca, tiene como requisito aumentar el tiempo de entrenamiento de la red para surtir efecto.

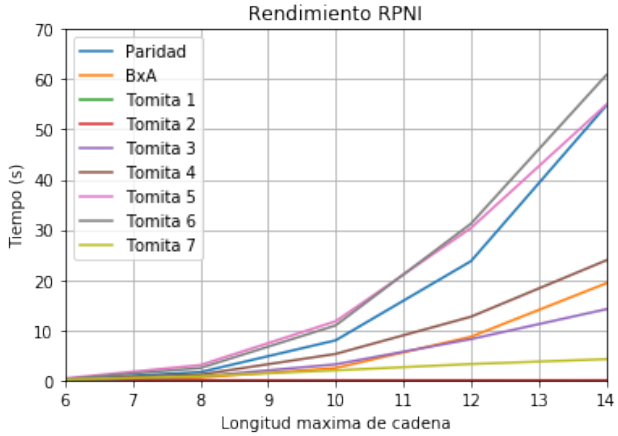


Figura 4.15: Tiempo de ejecución en segundos del algoritmo $RPNI$ para los diferentes problemas planteados frente a la longitud máxima de cadenas de entrenamiento.

Por otro lado, respecto a la cantidad de memoria utilizada por las diferentes técnicas, mientras que el algoritmo $LSTAR$ tiene un consumo trivial y las redes neuronales tienen una configuración variable, aunque normalmente más que aceptable, el algoritmo $RPNI$, debido al crecimiento exponencial del número de estados cuanto mayor es la longitud de las cadenas proporcionadas, aumenta también de forma exponencial la memoria consumida. Es por este motivo, fundamentalmente, que se hace inviable el uso del algoritmo $RPNI$ para problemas más complejos con un número más elevado de estados, no solo en tiempo de ejecución, sino también en la memoria que requiere el sistema.

Es por estas razones que la inferencia de gramáticas regulares utilizando un modelo de RNR que identifique correctamente los lenguajes objetivo, combinado con la extracción del autómata utilizando la red como oráculo, es la mejor alternativa a la hora de extraer el AFD a partir de un conjunto de datos de entrenamiento, ya que de esta forma se extrae directamente el autómata completo equivalente. Sin embargo, también se ha demostrado el más que interesante resultado de que, con un tiempo suficiente de entrenamiento y los valores apropiados de ruido, regularización y shock, la red parece estar implementando internamente el autómata.

Conclusiones

El objetivo de este trabajo ha sido realizar un estudio de las principales alternativas de inferencia de gramáticas regulares utilizando diferentes algoritmos clásicos y enfrentarlos al nuevo campo de investigación de inferencia de gramáticas utilizando aprendizaje profundo. Además, se han aportado ciertas mejoras que permiten a la red neuronal superar los problemas de generalización, sobre-regularización y evitar la cuantización del espacio de estados, además de proporcionar una nueva técnica para poder interpretar su comportamiento en la capa oculta y demostrar que la red podría estar implementando un autómata finito determinista equivalente.

Siguiendo estos objetivos, se ha presentado en el capítulo 2 el estado del arte en el campo de la inferencia de gramáticas regulares, haciendo un estudio tanto de los algoritmos clásicos como de las nuevas técnicas que se proponen en esta materia. En el capítulo (3) se ha presentado el diseño de pruebas, algoritmos, arquitectura de la red y metodología de análisis de rendimiento del conjunto. Por último, en el capítulo 4 se han presentado todos los resultados, con sus respectivas pruebas y evidencias, haciendo énfasis tanto en la eficacia como en la interpretabilidad de las técnicas desarrolladas.

Respecto a la interpretabilidad de la red neuronal cuando se enfrenta a ciertos problemas de decisión, como es clasificar los lenguajes regulares, se ha demostrado en la sección 4.1 que las neuronas tienen una funcionalidad concreta y determinante en la codificación de la solución del problema, utilizando instrucciones sencillas para, en su conjunto, ser capaz de generar la respuesta esperada y construir internamente un autómata finito determinista equivalente al lenguaje regular dado. Con esta conclusión, es interesante aplicar el algoritmo $LSTAR$ para extraer el AFD de la red cuando se presenta como oráculo. Ya que se ha demostrado la equivalencia entre este nuevo tipo de red neuronal recurrente y los autómatas finitos deterministas, además de la perfección en la clasificación de los lenguajes regulares estudiados en este trabajo, no se puede decir que esta red identifique en el límite la clase de los lenguajes regulares, pero sí es posible afirmar que para los nueve lenguajes estudiados la red está implementando directamente el AFD equivalente.

Los resultados de este proyecto, que han sido enviados a dos conferencias científicas internacionales [8, 9], plantean un nuevo abanico de investigación interesante a la hora de introducir un nuevo medio de interpretación de los problemas de clasificación más complejos, ya que cabría esperar un comportamiento similar en las neuronas de la red que lo utilizan. Por otro lado, sería también interesante extender este tipo de redes a problemas de predicción y analizar su comportamiento, ya que podría esperarse que algunas neuronas tuviesen un comportamiento probabilístico, activándose más intensamente a lo largo del tiempo ante ciertos patrones. Por último, otra alternativa sería introducir las nuevas mejoras planteadas en este proyecto a otras redes más complejas, como son las LSTM o las GRU, con el fin de analizar en más detalle el comportamiento de dichas redes.

Bibliografía

[1] S. C. Reghizzi, *Formal Languages and Compilation*. Springer Publishing Company, Incorporated, 1 ed., 2009.

[2] N. Chomsky, "Three models for the description of language," *IRE Transactions on Information Theory*, vol. 2, pp. 113–124, 1956.

[3] A. Graves, A. Mohamed, and G. E. Hinton, "Speech recognition with deep recurrent neural networks," in *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, pp. 6645–6649, 2013.

[4] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures," *Neural Networks*, vol. 18, no. 5-6, pp. 602–610, 2005.

[5] A. Graves and J. Schmidhuber, "Offline handwriting recognition with multidimensional recurrent neural networks," in *Advances in Neural Information Processing Systems 21, Proceedings of the Twenty-Second Annual Conference on Neural Information Processing Systems, Vancouver, British Columbia, Canada, December 8-11, 2008*, pp. 545–552, 2008.

[6] F. A. Gers and J. Schmidhuber, "LSTM recurrent networks learn simple context-free and context-sensitive languages," *IEEE Trans. Neural Networks*, vol. 12, no. 6, pp. 1333–1340, 2001.

[7] D. Eck and J. Schmidhuber, "Learning the long-term structure of the blues," in *Artificial Neural Networks - ICANN 2002, International Conference, Madrid, Spain, August 28-30, 2002, Proceedings*, pp. 284–289, 2002.

[8] C. Oliva and L. Lago-Fernández, "Interpretability of recurrent neural networks trained on regular languages," in *15th International Work-Conference on Artificial Neural Networks, IWANN, accepted*, 2019.

[9] C. Oliva and L. Lago-Fernández, "On the interpretation of recurrent neural networks as finite state machines," in *28th International Conference on Artificial Neural Networks, ICANN, submitted*, 2019.

[10] S. C. Kleene, "Representation of events in nerve nets and finite automata," in *Automata Studies* (C. Shannon and J. McCarthy, eds.), pp. 3–41, Princeton, NJ: Princeton University Press, 1956.

[11] P. Linz, *An introduction to formal languages and automata (4. ed.)*. Jones and Bartlett Publishers, 2006.

[12] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal of Research and Development*, vol. 3, pp. 210–229, July 1959.

[13] J. R. Koza, F. H. Bennett, D. Andre, and M. A. Keane, *Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming*, pp. 151–170. Dordrecht: Springer Netherlands, 1996.

[14] D. Poole, A. Mackworth, and R. Goebel, *Computational Intelligence: A Logical Approach*. Oxford University Press, 1998.

[15] M. Nielsen, "Neural networks and deep learning." `http://neuralnetworksanddeeplearning.com`.

[16] H. Sompolinsky, "The theory of neural networks: The hebb rule and beyond," in *Heidelberg Colloquium on Glassy Dynamics* (J. L. van Hemmen and I. Morgenstern, eds.), (Berlin, Heidelberg), pp. 485–527, Springer Berlin Heidelberg, 1987.

[17] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[18] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 12, pp. 2121–2159, July 2011.

[19] M. D. Zeiler, "ADADELTA: an adaptive learning rate method," *CoRR*, vol. abs/1212.5701, 2012.

[20] J. L. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, no. 2, pp. 179–211, 1990.

[21] C. L. Giles, G. Z. Sun, H. H. Chen, Y. C. Lee, and D. Chen, "Higher order recurrent networks & grammatical inference," in *Proceedings of the 2Nd International Conference on Neural Information Processing Systems*, NIPS'89, (Cambridge, MA, USA), pp. 380–387, MIT Press, 1989.

[22] J. B. Pollack, "The induction of dynamical recognizers," *Machine Learning*, vol. 7, pp. 227–252, Sep 1991.

[23] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[24] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pp. 1724–1734, 2014.

[25] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. New York, NY, USA: Cambridge University Press, 2010.

[26] A. Clark, F. Coste, and L. Miclet, eds., *Grammatical Inference: Algorithms and Applications, 9th International Colloquium, ICGI 2008, Saint-Malo, France, September 22-24, 2008, Proceedings*, vol. 5278 of *Lecture Notes in Computer Science*, Springer, 2008.

[27] H. Siegelmann and E. Sontag, "On the computational power of neural nets," *J. Comput. Syst. Sci.*, vol. 50, pp. 132–150, Feb. 1995.

[28] H. T. Siegelmann, "Recurrent neural networks and finite automata," *Computational Intelligence*, vol. 12, pp. 567–574, 1996.

[29] C. L. Giles, C. B. Miller, D. Chen, G. Sun, H. Chen, and Y. Lee, "Extracting and learning an unknown grammar with recurrent neural networks," in *Advances in Neural Information Processing Systems 4, [NIPS Conference, Denver, Colorado, USA, December 2-5, 1991]*, pp. 317–324, 1991.

[30] H. Jacobsson, "Rule extraction from recurrent neural networks: A taxonomy and review," *Neural Computation*, vol. 17, no. 6, pp. 1223–1263, 2005.

[31] A. Cleeremans, D. Servan-Schreiber, and J. Mcclelland, "Finite state automata and simple recurrent networks," *Neural Computation - NECO*, vol. 1, pp. 372–381, 09 1989.

[32] C. W. Omlin and C. L. Giles, "Extraction of rules from discrete-time recurrent neural networks," *Neural Networks*, vol. 9, no. 1, pp. 41–52, 1996.

[33] Z. Zeng, R. M. Goodman, and P. Smyth, "Learning finite state machines with self-clustering recurrent networks," *Neural Computation*, vol. 5, no. 6, pp. 976–990, 1993.

[34] Q. Wang, K. Zhang, A. G. Ororbia II, X. Xing, X. Liu, and C. L. Giles, "An empirical evaluation of rule extraction from recurrent neural networks," *Neural Computation*, vol. 30, no. 9, 2018.

[35] M. Tomita, "Dynamic construction of finite automata from examples using hill-climbing," in *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, (Ann Arbor, Michigan), pp. 105–108, 1982.

[36] J. J. Michalenko, A. Shah, A. Verma, S. Chaudhuri, and A. B. Patel, "Finite automata can be linearly decoded from language-recognizing RNNs," in *International Conference on Learning Representations*, 2019.

[37] J. F. Kolen, "Fool's gold: Extracting finite state machines from recurrent network dynamics," in *Advances in Neural Information Processing Systems 6, [7th NIPS Conference, Denver, Colorado, USA, 1993]*, pp. 501–508, 1993.

[38] A. Karpathy, J. Johnson, and F. Li, "Visualizing and understanding recurrent networks," *CoRR*, vol. abs/1506.02078, 2015.

[39] "Tensorflow." `https://www.tensorflow.org/`.

[40] "Keras." `https://keras.io/`.

[41] A. Karpathy, "Minimal character-level language model with a vanilla recurrent neural network." `https://gist.github.com/karpathy/d4dee566867f8291f086`.

[42] J. B. Tenenbaum, V. de Silva, and J. C. Langford, "A global geometric framework for nonlinear dimensionality reduction," *Science*, vol. 290, no. 5500, p. 2319, 2000.

[43] G. Weiss, Y. Goldberg, and E. Yahav, "Extracting automata from recurrent neural networks using queries and counterexamples," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, pp. 5244–5253, 2018.

Apéndices

A

Algoritmo de retropropagación

Este algoritmo utiliza el descenso por gradiente de la función de coste para ajustar los pesos de la red en todas sus capas. Hay que tener en cuenta que las variables $Salidas$ y $Salidas'$ son, precisamente, dos listas de las matrices de los valores de la función de activación y su derivada, respectivamente, para cada una de las capas de la red. Se puede demostrar que una red multicapa con al menos una capa oculta es un aproximador universal cuando utiliza este algoritmo de aprendizaje.

input : Salidas de cada capa de la red (aplicando función de activación): $Salidas$
input : Derivada de las salidas: $Salidas'$
input : Salida de la red esperada: t
input : Matrices de pesos de cada capa: $Pesos$
output: Matrices de pesos de cada capa actualizadas: $Pesos$

1 *El bucle for excluye el primer cálculo $Delta$ por si no hay capas ocultas*;
2 $\delta \leftarrow$ (t − Salidas [-1]) ∗ Salidas' [-1];
3 **for** $i \leftarrow$ `Length` (Pesos) **to** 0 **do**
4 $\quad \delta_{New} \leftarrow$ ($\delta \times$ Pesos [i]) * Salidas' [i];
5 $\quad$ Pesos [i] $\leftarrow$ Pesos [i] + α * `Traspuesta` (δ) * Salidas [i];
6 $\quad \delta \leftarrow \delta_{New}$;
7 **end**
8 Pesos [0] $\leftarrow$ Pesos [0] + α * `Traspuesta` (δ) * Salidas [0];
9 **return** Pesos;

Algoritmo A.1: Retropropagación utilizando el descenso por gradiente.

B

Algoritmos de inferencia de gramáticas regulares

B.1. K-Testable

B.1.1. Descripción de un lenguaje K-Testable

Un lenguaje $k-testable$ es un subconjunto de la clase de los lenguajes regulares que cumple que dadas dos cadenas w y w' que tienen la misma terminación con los k-1 últimos caracteres, si $w \in L$ entonces $w' \in L$. Formalmente se define una máquina $k-testable$ como una quíntupla Z = (Σ, I, F, T, C) donde Σ es el alfabeto, $I \subseteq \Sigma^{k-1}$ es el conjunto de prefijos de longitud k-1, $F \subseteq \Sigma^{k-1}$ es el conjunto de sufijos de longitud k-1, $T \subseteq \Sigma^k$ es el conjunto de segmentos de tamaño k que pertenecen a las cadenas de la presentación y $C \subseteq \Sigma^{<k}$ es el conjunto de cadenas de longitud menor que k que pertenecen a la presentación. Dada una máquina Z, el lenguaje $k-testable$ $L(Z)$ se define como el lenguaje que acepta las cadenas w que cumplen la siguiente ecuación:

$$L(Z) = (I\Sigma^* \cap \Sigma^* F - \Sigma^*(\Sigma^k - T)\Sigma^*) \cup C \tag{B.1}$$

es decir, (i) algún segmento de T es una subcadena de w, la cual empieza por algún prefijo de I y termina por algún sufijo de F, o (ii) es una cadena que pertenece al conjunto C.

Conjunto	Valores	Descripción
Σ	{ a, b }	Alfabeto
I	{ ab, aa, ba }	Prefijos de longitud k-1 = 2
F	{ ab, ba }	Sufijos de longitud k-1 = 2
T	{ aab, aba, abb, baa, bab, bba }	Segmentos de longitud k = 3
C	{ ab }	Cadenas de S de longitud menor que k

Tabla B.1: Ejemplo de máquina k-testable con S = { ab, ababba, abbababab, baababa, aaba, baabaab } y k = 3

B.1.2. GeneraKTestable

El algoritmo *GeneraKTestable* forma una máquina K-Testable a partir de una presentación positiva de los datos, es decir, a partir de un conjunto de ejemplos de cadenas aceptadas por el lenguaje. La función de esta máquina consiste exclusivamente en identificar los prefijos, sufijos y cadenas de conexión de longitud K para poder generar un autómata que reconozca la gramática.

```
input : Presentación S
input : Valor K
output: Máquina K-testable: (Σ, I, F, T, C)
1  foreach w in S do
2      Σ ← GetChars( w );
3      if Length( w ) < K then C ← w;
4      if Length( w ) >= K-1 and K ! = 1 then
5          I ← w [:K-1];
6          F ← w [K-1:];
7      end
8      if Length( w ) > K then
9          for i ← 0 to Length( w ) - K do T ← w [i:i+K] ;
10     end
11 end
12 return ( Σ, I, F, T, C)
```

Algoritmo B.1: *GeneraKTestable* - Generar una máquina k-testable a partir de una presentación positiva.

B.1.3. KTestableToAFD

Este último algoritmo es capaz de generar a partir de una máquina K-Testable bien definida el autómata finito determinista correspondiente al lenguaje que está identificando. Nótese en el algoritmo la unión de símbolos y cadenas en las variables de los bucles, donde, por ejemplo, pau no es una variable, sino la concatenación de la cadena p con el símbolo a y la cadena u.

input : Máquina K-testable: (Σ, I, F, T, C)
output: AFD

1 *Se define el estado inicial*;
2 $\mathsf{I} \leftarrow q_\lambda$;
3 *Se definen los estados del autómata*;
4 **foreach** pu *in* `Union(` I, C `)` *donde* p, u *son cadenas* **do** $\mathsf{Q} \leftarrow q_p$;
5 **foreach** au *in* T *donde* a *es un símbolo y* u *una cadena* **do** $\mathsf{Q} \leftarrow q_u$;
6 **foreach** ua *in* T *donde* a *es un símbolo y* u *una cadena* **do** $\mathsf{Q} \leftarrow q_u$;
7 *Se definen las transiciones del autómata*;
8 **foreach** pau *in* `Union(` I, C `)` *donde* a *es un símbolo y* p, u *son cadenas* **do** $\delta \leftarrow <q_p \rightarrow_a q_{pa}>$;
9 **foreach** aub *in* T *donde* a, b *son símbolos y* u *es una cadena* **do** $\delta \leftarrow <q_{au} \rightarrow_b q_{ub}>$;
10 *Se definen los estados finales del autómata*;
11 **foreach** w *in* `Union(` F, C `)` **do** $\mathsf{F} \leftarrow q_w$;
12 **return** `CreaAFD(` I, Q, δ, F `)`

Algoritmo B.2: *KTestableToAFD* - Construir un AFD a partir de una máquina k-testable.

B.2. RPNI

B.2.1. RPNI

El algoritmo $RPNI$ genera a partir de un informante (conjunto de ejemplos positivos y negativos de un lenguaje concreto) el autómata finito determinista mínimo equivalente cuando existe un número suficiente de cadenas con longitud suficiente. El mecanismo consiste, básicamente, en ir disminuyendo el número de estados del PTA inicial comprobando que los datos siguen siendo consistentes, es decir, las cadenas que pertenecen al lenguaje siguen siendo aceptadas y las cadenas negativas siguen siendo rechazadas. Nótese que aunque el algoritmo devuelva un PTA, este puede ser considerado un AFD si todos los estados de $rechazo$ se transforman en un estado normal.

```
   input : Informante: (S+, S−)
   output: Autómata Finito Determinista: AFD
1  PTA ← CONSTRUIR-PTA( (S+, S−) );
2  RED ← PTAGetEstadoInicial( PTA );
3  foreach s in PTAGetAlfabeto( PTA ) do BLUE ← q_s ;
4  while not Empty( BLUE ) do
5      Q_BLUE ← Pop( BLUE );
6      foreach Q_RED in RED do
7          PTAMerged ← MERGE( PTA, Q_RED, Q_BLUE );
8          if COMPATIBLE( PTAMerged, S− ) then
9              PTA ← PTAMerged;
10             foreach Q_aux in RED do BLUE ← PTAGetSucesores( PTA, Q_aux );
11         else
12             RED, BLUE ← PROMOCIONAR( Q_BLUE, RED, BLUE,
                PTAGetAlfabeto( PTA ), PTAGetFuncionTransicion( PTA ) );
13         end
14     end
15 end
16 Marca los estados S− in RED como no finales;
17 foreach w in S− do PTASetNoFinal( PTA, RED, S− );
18 En este punto PTA es un AFD;
19 return PTA
```

Algoritmo B.3: *RPNI* - obtener un AFD a partir de un informante.

B.2.2. Construir-PTA

input : Presentación: (S_+, S_-)
output: Máquina de estados PTA

1 *Se define el estado inicial*;
2 I $\leftarrow q_\lambda$;
3 *Se definen los estados del autómata*;
4 **foreach** pu *in* `Union(` S_+, S_- `)` *donde* p, u *son cadenas* **do** Q $\leftarrow q_p$;
5 *Se definen las transiciones del autómata*;
6 **foreach** pau *in* `Union(` S_+, S_- `)` *donde* a *es un símbolo y* p, u *son cadenas* **do** $\delta \leftarrow$ q_p transita con a al estado q_{pa} ;
7 *Se definen los estados finales del autómata*;
8 **foreach** w *in* S_+ **do** F $\leftarrow q_w$;
9 *Se definen los estados de rechazo del autómata*;
10 **foreach** w *in* S_- **do** R $\leftarrow q_w$;
11 **return** `CreaPTA(` I, Q, δ, F, R `)`

Algoritmo B.4: *RPNI ConstruirPTA* - Construir un PTA a partir de un informante (S_+, S_-).

B.2.3. Promocionar

input : Estado a promocionar: Q
input : Lista de estados RED: RED
input : Lista de estados BLUE: BLUE
input : Alfabeto: Σ
input : Función de transición: δ
output: Lista de estados RED actualizada: RED
output: Lista de estados BLUE actualizada: BLUE

1 *Se actualiza la lista de estados BLUE con todos los sucesores de Q*;
2 **foreach** s *in* Σ **do** BLUE $\leftarrow \delta$ (Q, s) ;
3 RED $\leftarrow Q$;
4 **return** RED, BLUE

Algoritmo B.5: *RPNI Promocionar* - Transformar un estado de $BLUE$ a RED.

B.2.4. Compatible

```
input : Autómata PTA: PTA
input : Cadenas rechazadas: S_
output: Boolean
1 foreach w in S_ do
2     if PTAAcepta( PTA, w ) then return False;
3 end
4 return True
```

Algoritmo B.6: *RPNI Compatible* - Comprobar la consistencia del autómata con el informante.

B.2.5. Fold

```
input : Autómata PTA: PTA
input : Estado de RED: q_RED
input : Estado de BLUE: q_BLUE
output: Autómata PTA actualizado: PTA
1  foreach s in PTAGetAlfabeto( PTA ) do
2      Q_B ← PTATransitaConSimbolo( PTA, q_BLUE, s );
3      if ( Q_R ← PTATransitaConSimbolo( PTA, q_RED, s ) ) != NULL then
4          FOLD( PTA, Q_R, Q_B );
5      end
6      else
7          PTAAddTransicion( PTA, q_RED, s, Q_B );
8      end
9  end
10 return PTA
```

Algoritmo B.7: *RPNI Fold* - Función auxiliar para unir dos estados del autómata y sus respectivos sucesores.

B.2.6. Merge

input : Autómata PTA: PTA
input : Estado de RED: q_{RED}
input : Estado de BLUE: q_{BLUE}
output: Autómata PTA actualizado: PTA

```
1 foreach q in PTA do
2     foreach s in PTAGetAlfabeto( PTA ) do
3         if q_BLUE = PTATransitaConSimbolo( PTA, q, s ) then
4             PTAAddTransicion( PTA, q, s, q_RED );
5             PTABorraTransicion( PTA, q, s, q_BLUE );
6         end
7     end
8 end
9 return FOLD( PTA, q_RED, q_BLUE )
```

Algoritmo B.8: *RPNI Merge* - Unir dos estados del autómata.

B.3. LSTAR

B.3.1. LSTAR

El algoritmo $LSTAR$ genera una tabla de observación (OT) a partir de consultas externas que va realizando al oráculo para, por último, transformar la tabla en un AFD equivalente. El algoritmo, cuya rutina principal es bastante sencilla, consiste en obtener una OT cerrada y consistente para luego ejecutar una consulta al oráculo en espera de un contraejemplo que añadir a la OT. Cuando no existe ningún contraejemplo, el algoritmo ha finalizado y el autómata objetivo se puede generar.

```
input : Oracle: oracle
input : Alfabeto: Σ
input : Presentación: S = <S+, S->
output: Autómata Finito Determinista: AFD
1  OT ← LSTARInicializar( oracle, Σ );
2  while Answer != YES do
3      while not Cerrada( OT ) or not Consistente( OT ) do
4          if not Cerrada( OT ) then OT ← LSTARCerrar( OT, oracle, Σ );
5          if not Consistente( OT ) then OT ← LSTARConsistente( OT,
           oracle, Σ );
6      end
7      Answer ← EQ( S );
8      if Answer != YES then OT ← LSTARUsarEQ( OT, oracle, Σ, S );
9  end
10 return LSTARCreaAutomata( OT, Σ );
```

Algoritmo B.9: LSTAR.

B.3.2. CreaAutomata

```
input : Tabla de observación: OT = <C, E, S, R>
input : Alfabeto: Σ
output: Autómata equivalente: DFA
1  PTA ← CreaPTA();
2  OTCompleto ← Union( OT [RED], OT [BLUE] );
3  Se crean los estados del autómata;
4  foreach e_RED in OT [RED] do
5      if OTCompleto [e_RED].R != OTCompleto [e_PTA].R for all e_PTA in
       GetEstados( PTA ) then AddEstado( PTA, e_RED );
6  end
7  Se definen estados finales;
8  foreach e_PTA in GetEstados( PTA ) do
9      if OTCompleto [e_PTA,λ] = 1 then SetEstadoFinal( PTA, e_PTA );
10 end
11 Se generan las transiciones;
12 foreach e_PTA in GetEstados( PTA ) do
13     foreach simbolo in Σ do
14         if OTCompleto [e_PTA·simbolo].R = OTCompleto [eDest_PTA].R for any
           eDest_PTA in GetEstados( PTA ) then AddTransicion( PTA, e_PTA,
           simbolo, eDest_PTA );
15     end
16 end
17 return PTA;
```

Algoritmo B.10: *LSTAR CreaAutomata* - Genera el AFD correspondiente a la tabla de observación proporcionada.

B.3.3. Cerrar

```
input : Tabla de observación: OT = <C, E, S, R>
input : Oracle: oracle
input : Alfabeto: Σ
output: Tabla de observación actualizada: OT = <C, E, S, R>
1  OTCompleto ← Union( OT [RED], OT [BLUE] );
2  foreach e_BLUE in OT [BLUE] do
3      if OTCompleto [e_BLUE] != OTCompleto [e_RED] for all e_RED in OT [RED] then
4          OT [RED] = Pop( OT [BLUE], e_BLUE );
5          foreach simbolo in Σ do
6              OT [BLUE]← e_BLUE·simbolo;
7              OT [BLUE, e_BLUE·simbolo]← MQ( oracle, e_BLUE·simbolo·e ) for each e in OT.S;
8          end
9      end
10 end
11 return OT;
```

Algoritmo B.11: *LSTAR Cerrar* - Cierra la tabla de observación dada.

B.3.4. Inicializar

```
input : Oracle: oracle
input : Alfabeto: Σ
output: Tabla de observación: OT = <C, E, S, R>
1  OT.S ← λ;
2  OT [RED]← λ;
3  OT [RED, λ] ← MQ( oracle, λ );
4  foreach simbolo in Σ do
5      OT [BLUE]← simbolo;
6      OT [BLUE, simbolo] ← MQ( oracle, simbolo );
7  end
8  return OT;
```

Algoritmo B.12: *LSTAR Inicializar* - Genera la tabla de observación inicial para el algoritmo $LSTAR$.

B.3.5. Consistente

input : Tabla de observación: OT = $<C, E, S, R>$
input : Oracle: $oracle$
input : Alfabeto: Σ
output: Tabla de observación actualizada: OT = $<C, E, S, R>$

```
1  OTCompleto ← Union( OT [RED], OT [BLUE] );
2  foreach e1 in OT [RED] do
3      if OTCompleto [e1] = OTCompleto [e2] for each e2 in OT [RED] then
4          foreach a in Σ do
5              if OTCompleto [s1·a, e] != OTCompleto [s2·a, e] for each e in OT.R then
6                  OT.R← a·e;
7                  foreach estado in OTCompleto do
8                      OTCompleto [estado]← MQ ( oracle, estado·a·e );
9                  end
10             end
11         end
12     end
13 end
14 return OT;
```

Algoritmo B.13: *LSTAR Consistente* - Intenta hacer consistente la tabla de observación añadiendo un nuevo elemento a S.

B.3.6. UsarEQ

```
input : Tabla de observación: OT = <C, E, S, R>
input : Oracle: oracle
input : Alfabeto: Σ
input : Respuesta: answer
output: Tabla de observación actualizada: OT = <C, E, S, R>
1  OTCompleto ← Union( OT [RED], OT [BLUE] );
2  P ← λ;
3  for c ← 0 to Length( answer ) +1 do
4      P ← answer[:c];
5      if P not in OT [RED] then
6          OT [RED]← P;
7          OT [RED, P ]← MQ( oracle, P ·e ) for each e in OT.S;
8          if P ·a not in answer[:c+1] for each a in Σ then
9              if P ·a not in OT [BLUE] then
10                 OT [BLUE]← P ·a;
11                 OT [BLUE, P ·a]← MQ( oracle, P ·a·e ) for each e in OT.S;
12             end
13         end
14     end
15     if P in OT [BLUE] then Pop( OT [BLUE], P );
16 end
17 return OT;
```

Algoritmo B.14: *LSTAR UsarEQ* - A partir de la respuesta dada por la consulta EQ, actualiza la tabla de observación.

C

Implementación de la RNR

Este apéndice contiene el código de la red neuronal recurrente utilizado a lo largo de este proyecto. Las mejoras aplicadas pueden ser consultadas en (1) la función $lossFun$ (código C.4), donde se introduce el ruido gausiano a la función de activación y (2) la función $train$ (código C.10, C.11 y C.12) donde se añaden los conceptos de shock y regularización. Nótese que el software también tiene preparado el código del optimizador Adagrad, aunque finalmente no se utiliza, y se ha añadido la regularización L2 pero no tiene ningún efecto, por lo que durante toda la fase de experimentos el valor de regularización L2 es cero.

Código C.1: Copyright del autor del código inicial [41].

```
0  # -*-coding: utf-8 -*-
1  """
2  Minimal character-level Vanilla RNN model. Written by Andrej Karpathy (@karpathy)
3  BSD License
4  """
```

Código C.2: Implementación auxiliar de las funciones de activación típicas.

```
18 def sigmoid(x):
19     return 1. / (1. + np.exp(-x))
20
21 def derivada_sigmoid(x):
22     return x*(1 -x)
23
24 def tanh(x):
25     return np.tanh(x)
26
27 def derivada_tanh(x):
28     return (1 -x*x)
```

C.1. Constructor

Código C.3: Constructor de la clase RNN.

```
32      def __init__(self, input_file, output_file, hidden_size, noise_level=0., seq_length=25,
            learning_rate=1e-1, activacion="TANH"):
33          self.data = open(input_file, 'r').read()
34          self.chars = list(set(self.data))
35          self.data_size, self.vocab_size = len(self.data), len(self.chars)
36          print 'data␣INPUT␣has␣%d␣characters,␣%d␣unique.' % (self.data_size, self.vocab_size)
37          self.char_to_ix = { ch:i for i,ch in enumerate(self.chars) }
38          self.ix_to_char = { i:ch for i,ch in enumerate(self.chars) }
39
40          self.data_y = open(output_file, 'r').read()
41          self.chars_y = list(set(self.data_y))
42          self.data_y_size, self.vocab_y_size = len(self.data_y), len(self.chars_y)
43          print 'data␣OUTPUT␣has␣%d␣characters,␣%d␣unique.' % (self.data_y_size, self.vocab_y_size)
44          self.char_to_ix_y = { ch:i for i,ch in enumerate(self.chars_y) }
45          self.ix_to_char_y = { i:ch for i,ch in enumerate(self.chars_y) }
46
47          # hyperparameters
48          self.hidden_size = hidden_size # size of hidden layer of neurons
49          self.seq_length = seq_length # number of steps to unroll the RNN
50          self.learning_rate = learning_rate # learning rate
51
52          # model parameters
53          self.Wxh = np.random.randn(self.hidden_size, self.vocab_size)*0.01 # input to hidden weights
                matrix
54          self.Whh = np.random.randn(self.hidden_size, self.hidden_size)*0.01 # hidden to hidden weights
                matrix
55          self.Why = np.random.randn(self.vocab_y_size, self.hidden_size)*0.01 # hidden to output weights
                matrix
56
57          self.bh = np.zeros((self.hidden_size, 1)) # hidden bias
58          self.by = np.zeros((self.vocab_y_size, 1)) # output bias
59
60          self.noise_level = noise_level
61          self.h = 0
62          self.n = 0
63
64          self.activacion = activacion
65          self.defineActivacion(activacion)
```

C.2. Función de coste

Código C.4: Función de coste y backpropagation (I).

```
82      def lossFun(self, inputs, targets, hprev):
83          """
84          inputs,targets are both list of integers.
85          hprev is Hx1 array of initial hidden state
86          returns the loss, gradients on model parameters, and last hidden state
87          """
88          xs, hs, ys, ps = {}, {}, {}, {}
89          hs[-1] = np.copy(hprev)
90          loss = 0
91          # forward pass
92          for t in xrange(len(inputs)):
93              xs[t] = np.zeros((self.vocab_size,1)) # encode in 1-of-k representation
94              xs[t][inputs[t]] = 1
95
96              ################################## IWANN ---NOISE
                    ##################################################
97              hs[t] = self.f_activacion(np.dot(self.Wxh, xs[t]) +
98                                        np.dot(self.Whh, hs[t-1]) +
99                                        self.bh +
100                                       self.noise_level *np.random.randn(self.hidden_size,1) *hs[t-1])
101             ################################## IWANN ---NOISE
                    ##################################################
102
103             ys[t] = np.dot(self.Why, hs[t]) + self.by # unnormalized log probabilities for next chars
104             ps[t] = np.exp(ys[t]) / np.sum(np.exp(ys[t])) # probabilities for next chars
105             loss += -np.log(ps[t][targets[t],0]+1e-8) # softmax (cross-entropy loss)
106
107         ######################### IWANN ---Regularization
                ###################################
108         loss +=
                self.l1reg*(np.sum(np.abs(self.Wxh))+np.sum(np.abs(self.Whh))+np.sum(np.abs(self.Why)))
109         loss += self.l2reg*(np.sum(self.Wxh**2)+np.sum(self.Whh**2)+np.sum(self.Why**2))
110         ######################### IWANN ---Regularization
                ###################################
```

Código C.5: Función de coste y backpropagation (II).

```
112         # backward pass: compute gradients going backwards
113         dWxh, dWhh, dWhy = np.zeros_like(self.Wxh), np.zeros_like(self.Whh), np.zeros_like(self.Why)
114         dbh, dby = np.zeros_like(self.bh), np.zeros_like(self.by)
115         dhnext = np.zeros_like(hs[0])
116         for t in reversed(xrange(len(inputs))):
117             dy = np.copy(ps[t])
118             dy[targets[t]] -= 1 # backprop into y. see
                    http://cs231n.github.io/neural-networks-case-study/#grad if confused here
119             dWhy += np.dot(dy, hs[t].T)
120             dby += dy
121             dh = np.dot(self.Why.T, dy) + dhnext # backprop into h
122             dhraw = self.f_deriv_activacion(hs[t]) *dh # backprop through f_activacion nonlinearity
123             dbh += dhraw
124             dWxh += np.dot(dhraw, xs[t].T)
125             dWhh += np.dot(dhraw, hs[t-1].T)
126             dhnext = np.dot(self.Whh.T, dhraw)
127         for dparam in [dWxh, dWhh, dWhy, dbh, dby]:
128             np.clip(dparam, -5, 5, out=dparam) # clip to mitigate exploding gradients
129         return loss, dWxh, dWhh, dWhy, dbh, dby, hs[len(inputs)-1]
```

C.3. Entrenamiento

Código C.6: Función de entrenamiento de la red (I).

```
190     def train(self, limit=100000, limit_cost=1e-3, shock=0.5, check_shock=5000, l1reg=0, l2reg=0,
            verbose=False, validate=False):
191         p = 0
192         self.l1reg = l1reg
193         self.l2reg = l2reg
194         mWxh, mWhh, mWhy = np.zeros_like(self.Wxh), np.zeros_like(self.Whh), np.zeros_like(self.Why)
195         mbh, mby = np.zeros_like(self.bh), np.zeros_like(self.by) # memory variables for Adagrad
196         error = True
197
198         if self.n == 0:
199             self.c_aprendizaje = []
200             self.c_loss = []
```

Código C.7: Función de entrenamiento de la red (II).

```
201             n = 0
202             smooth_loss = -np.log(1.0/self.vocab_size)*self.seq_length
203         else:
204             n = self.n
205             smooth_loss = self.smooth_loss
206
207         print "␣>>␣Entrenando..."
208         ################################## ICANN ---SHOCK
                ##################################
209         smooth_loss_prev = 0 # used to see if we should shock or not the neurons
210         num_encendidas = 0 # number of activated neurons
211         limit_ini = limit # we use the SUM(1/2^n)
212         limit_percentaje = 1.0 # this variable store the 2^n each time
213         ################################## ICANN ---SHOCK
                ##################################
214         while n < limit+1 and smooth_loss > limit_cost and error:
215             self.n = n
216             self.smooth_loss = smooth_loss
217             # prepare inputs (we're sweeping from left to right in steps seq_length long)
218             if p+self.seq_length+1 >= len(self.data) or n == 0:
219                 self.h = np.zeros((self.hidden_size,1)) # reset RNN memory
220                 p = 0 # go from start of data
221             inputs = [self.char_to_ix[ch] for ch in self.data[p:p+self.seq_length]]
222             targets = [self.char_to_ix_y[ch] for ch in self.data_y[p:p+self.seq_length]]
223
224             # forward seq_length characters through the net and fetch gradient
225             loss, dWxh, dWhh, dWhy, dbh, dby, self.h = self.lossFun(inputs, targets, self.h)
226
227             smooth_loss = smooth_loss *0.999 + loss *0.001
228
229             if n % 1000 == 0:
230                 if verbose: print '␣>>␣iter␣%d,␣loss:␣%f' % (n, smooth_loss)
231                 self.c_loss.append([n, smooth_loss])
232
233                 ################################## ICANN ---SHOCK
                        ##################################
234                 #
235                 # Se producira un shock cuando haya alguna neurona apagada y se considere oportuno.
236                 #
237                 hsaux, _, _ = self.estados(muestra=50)
238                 num_encendidas = len(hsaux[:, np.where(np.mean(np.abs(hsaux), axis=0)>0.3)].T)
239
240                 if n % check_shock == 0 and smooth_loss > self.seq_length*0.04:
```

Código C.8: Función de entrenamiento de la red (III).

```
241                     if n != 0 and smooth_loss > smooth_loss_prev and num_encendidas <
                            self.hidden_size:
242                         if verbose: print "___Number_of_active_neurons:_{0}".format(num_encendidas)
243                         self.Wxh += np.random.randn(np.shape(self.Wxh)[0],np.shape(self.Wxh)[1]) *shock
244                         self.Whh += np.random.randn(np.shape(self.Whh)[0],np.shape(self.Whh)[1]) *
                                shock
245                         self.Why += np.random.randn(np.shape(self.Why)[0],np.shape(self.Why)[1]) *shock
246                         if verbose: print "___SHOCK"
247                         limit += limit_ini*limit_percentaje
248                         limit_percentaje *= 0.5
249                         if verbose: print "___Setting_new_limit_to_{0}_iters...".format(limit)
250                     smooth_loss_prev = smooth_loss
251                 ################################## ICANN ---SHOCK
                        ##################################
252
253                 if n != 0 and verbose and n % 10000 == 0:
254                     print "___Number_of_active_neurons:_{0}".format(num_encendidas)
255
256                 if n != 0 and validate and n % 10000 == 0:
257                     print "___Validating..."
258                     validationLen = 50000
259                     pError, _ = self.validate_string(self.data[:validationLen], self.data_y[:validationLen])
260                     print "_____" + str(pError) + "%_accuracy"
261                     if pError == 100: break
262
263             # perform parameter update with Adagrad
264             for param, dparam, mem in zip([self.Wxh, self.Whh, self.Why, self.bh, self.by],
265                                           [dWxh, dWhh, dWhy, dbh, dby],
266                                           [mWxh, mWhh, mWhy, mbh, mby]):
267                 mem += dparam *dparam
268                 #param += -self.learning_rate *dparam / np.sqrt(mem + 1e-8) # adagrad update
269                 param += -self.learning_rate *dparam
270
271
272             ############################## IWANN ---Regularizacion L1
                    ###################################
273             self.Why -= np.sign(self.Why)*l1reg
274             self.Whh -= np.sign(self.Whh)*l1reg
275             self.Wxh -= np.sign(self.Wxh)*l1reg
276             ############################## IWANN ---Regularizacion L1
                    ###################################
277
278             ############################## IWANN ---Regularizacion L2
                    ###################################
279             self.Why -= self.Why*l2reg
280             self.Whh -= self.Whh*l2reg
```

Código C.9: Función de entrenamiento de la red (IV).

```
281             self.Wxh -= self.Wxh*l2reg
282             ############################ IWANN ---Regularizacion L2
                    ################################
283
284             p += self.seq_length # move data pointer
285             n += 1 # iteration counter
286
287         print "----------DONE----------"
```

C.4. Funciones auxiliares

Código C.10: Función de contador de errores.

```
131     def errores(self, inputs_ok, targets_ok):
132         h = self.h
133         errores = 0
134         cadenas_errores = []
135         cadena = '$'
136         error = False
137
138         for i, t in enumerate(inputs_ok):
139             if t == 2:
140                 if error:
141                     cadenas_errores.append(cadena)
142                 cadena = '$'
143                 error = False
144             else:
145                 cadena += self.ix_to_char[t]
146
147             x = np.zeros((self.vocab_size, 1))
148             x[t] = 1
149             h = self.f_activacion(np.dot(self.Wxh, x) + np.dot(self.Whh, h) + self.bh)
150             y = np.dot(self.Why, h) + self.by
151             p = np.exp(y) / np.sum(np.exp(y))
152             ix = np.where(np.max(p) == p.ravel())[0][0]
153             if ix != targets_ok[i]:
154                 errores += 1
155                 error = True
156         return errores + 0., cadenas_errores
```

Código C.11: Función de validación de un string.

```
158     def validate_string(self, inputs, targets):
159         """
160         validate a sequence of integers from the model
161         """
162         h = self.h
163         ixes = []
164         inputs_ok = [self.char_to_ix[ch] for ch in inputs]
165         targets_ok = [self.char_to_ix_y[ch] for ch in targets]
166         pError, cadenas_errores = self.errores(inputs_ok, targets_ok)
167         return 100 -pError/len(inputs) *100, cadenas_errores
```

Código C.12: Función de test.

```
169     def test(self, test_input, test_output):
170         pError, cadenas_errores = self.validate_string(open(test_input, 'r').read(), open(test_output,
                'r').read())
171         return 100 -pError, cadenas_errores
```

Código C.13: Función de comprobación de aceptación de una cadena.

```
173     def acepta(self, inputs):
174         """
175         validate a sequence of integers from the model and return if last prediction is 1
176         """
177         h = self.h
178         ixes = []
179         inputs_ok = [self.char_to_ix[ch] for ch in inputs]
180 
181         for t in inputs_ok:
182             x = np.zeros((self.vocab_size, 1))
183             x[t] = 1
184             h = self.f_activacion(np.dot(self.Wxh, x) + np.dot(self.Whh, h) + self.bh)
185             y = np.dot(self.Why, h) + self.by
186             p = np.exp(y) / np.sum(np.exp(y))
187             ix = np.where(np.max(p) == p.ravel())[0][0]
188         return self.ix_to_char_y[ix] == "1"
```

Código C.14: Función para obtener el mapa de colores (I).

```
349     def pinta_strings(self, inputs_ini=None, umbral=0., print_acc=False, name='Acc'):
350         if inputs_ini == None:
351             estados, probs, inputs = self.estados(offset=500, muestra=60, por_Whh=False)#,
                    inputs=inputs_ini)
352         else:
353             estados, probs, inputs = self.estados(offset=500, muestra=60, por_Whh=False,
                    inputs=inputs_ini)
354 
355         inputs_print = ''.join([self.ix_to_char[x] for x in inputs])
356 
357         # Accept or not
358         if print_acc:
359             fig, ax = plt.subplots(figsize=(20,20))
360             plt.ylabel(name, rotation=0, position=(-8,0), fontsize=18)
361             #q.set_rotation(0, fontsize=20, labelpad=20)
362             for i, simbolo in enumerate(inputs_print):
363                 if probs[i] > 0.5:
364                     rectangle = mpatch.Rectangle((i,0), 1, 1, color=(1.,0.8,0.))
365                 else:
366                     rectangle = mpatch.Rectangle((i,0), 1, 1, color=(1.,1.,1.))
367 
368                 ax.add_artist(rectangle)
369                 rx, ry = rectangle.get_xy()
370                 cx = rx + rectangle.get_width()/2.0
371                 cy = ry + rectangle.get_height()/2.0
372 
373                 ax.annotate(simbolo, (cx, cy), color='k', weight='bold',
374                             fontsize=14, ha='center', va='center')
375 
376             ax.set_aspect('equal')
377             ax.set_xlim((0, 60))
378             plt.xticks([])
379             plt.yticks([])
380             plt.show()
381 
382         # Every neuron
383         for neurona in range(self.hidden_size):
384             if np.mean(np.abs(estados[:, neurona])) < umbral or np.mean(estados[:, neurona]) > 1-umbral
                    or np.mean(estados[:, neurona]) < -1+umbral:
385                 continue
386             fig, ax = plt.subplots(figsize=(20,20))
387             plt.ylabel(str(neurona))
388             for i, simbolo in enumerate(inputs_print):
```

Código C.15: Función para obtener el mapa de colores (II).

```
389                 if estados[i][neurona] < 0:
390                     rectangle = mpatch.Rectangle((i,0), 1, 1, color=(1.+estados[i][neurona],
391                                                                      1.,
392                                                                      1.))
393                 else:
394                     rectangle = mpatch.Rectangle((i,0), 1, 1, color=(1.,
395                                                                      1.-estados[i][neurona],
396                                                                      1.-estados[i][neurona]))
397
398                 ax.add_artist(rectangle)
399                 rx, ry = rectangle.get_xy()
400                 cx = rx + rectangle.get_width()/2.0
401                 cy = ry + rectangle.get_height()/2.0
402
403                 ax.annotate(simbolo, (cx, cy), color='k', weight='bold',
404                             fontsize=14, ha='center', va='center')
405
406             ax.set_aspect('equal')
407             ax.set_xlim((0, 60))
408             plt.xticks([])
409             plt.yticks([])
410             plt.show()
411         return estados, probs, inputs
```

Código C.16: Función de pintado de la proyección Isomap.

```
451     def pintador_isomap(self, numSimb=1000, separador=0.0):
452         hs, probs, inputs = self.estados_isomap(numSimb=numSimb)
453         t1 = transform.Transformador(hs).isomap(n_neighbors=numSimb/2)
454         t1 += np.random.randn(np.shape(t1)[0], np.shape(t1)[1]) *separador
455         p1 = pint.Pintador(t1, inputs, self.vocab_size, probs=probs)
456         p1.pinta_clusters(figsize=(6,6))
457         return p1
```

Código C.17: Función para obtener estados de cadenas de aceptación.

```
413     def estados_isomap(self, numSimb=1000):
414         h = self.h
415         hs = []
416         maybeHs = []
417         maybeProbs = []
418         maybeInputs = []
419         probs = []
420         ix = self.data.index('$')
421         inputs_ok = [self.char_to_ix[ch] for ch in self.data[ix:]]
422         l = 0
423         inputs_final = []
424 
425         while l < numSimb:
426             ix += 1
427             if ix == len(inputs_ok): ix = self.data.index('$') # Por si da la vuelta al fichero completo (no
                    creo que pase)
428 
429             t = inputs_ok[ix]
430             if self.ix_to_char[t] == "$":
431                 if maybeProbs[-1]:
432                     hs += maybeHs
433                     probs += maybeProbs
434                     inputs_final += maybeInputs
435                     l += len(maybeInputs)
436                 maybeHs = []
437                 maybeProbs = []
438                 maybeInputs = []
439 
440             x = np.zeros((self.vocab_size, 1))
441             x[t] = 1
442             h = self.f_activacion(np.dot(self.Wxh, x) + np.dot(self.Whh, h) + self.bh)
443             y = np.dot(self.Why, h) + self.by
444             p = np.exp(y) / np.sum(np.exp(y))
445             maybeHs.append(np.transpose(h)[0])
446             maybeProbs.append(self.ix_to_char_y[np.where(np.max(p) == p.ravel())[0][0]] == "1")
447             maybeInputs.append(t)
448 
449         return np.array(hs), np.array(probs), np.array(inputs_final)
```

Código C.18: Función para obtener estados internos de la red.

```
321     def estados(self, offset=1000, muestra=500, por_Whh=False, inputs=None):
322         h = self.h
323         hs = []
324         probs = []
325         ix = self.data.index('$', offset)
326         #ix = offset
327         if inputs == None:
328             inputs_ok = [self.char_to_ix[ch] for ch in self.data[ix:ix+muestra]]
329         else:
330             inputs_ok = [self.char_to_ix[ch] for ch in inputs]
331 
332         for t in inputs_ok:
333             x = np.zeros((self.vocab_size, 1))
334             x[t] = 1
335             h = self.f_activacion(np.dot(self.Wxh, x) + np.dot(self.Whh, h) + self.bh)
336             y = np.dot(self.Why, h) + self.by
337             p = np.exp(y) / np.sum(np.exp(y))
338             ix = np.where(np.max(p) == p.ravel())[0][0]
339             hs.append(np.transpose(h)[0])
340             probs.append(p.ravel()[0])
341 
342         if por_Whh: return np.array([np.dot(self.Whh, h) for h in hs]), np.array(probs),
                np.array(inputs_ok)
343         else: return np.array(hs), np.array(probs), np.array(inputs_ok)
```


UAM
UNIVERSIDAD AUTONOMA
DE MADRID